Chapter

An Efficient Region Merging Algorithm in Raster Space

Borut Žalik, David Podgorelec, Niko Lukač, Krista Rizman Žalik and Domen Mongus

Abstract

This work introduces a new region merging algorithm operating in raster space represented by a 4-connected graph. Necessary definitions are introduced first to derive a new merging function formally. An implementation is described after that, which consists of two steps: a determination of the shared trails of the input cycles, and construction of the resulting merged region. The cycles defining the regions are represented by the Freeman crack chain code in four directions. The algorithm works in linear time $O(n)$, where n is the number of total graph vertices, i.e. pixels. However, the expected time complexity for one merging operation performed by the algorithm is $O(1)$.

Keywords: computer science, algorithms, 4-connected graph, merging function, chain code

1. Introduction

Region merging is one of the most commonly performed tasks in image processing that enables Object-Based Image Analysis (OBIA). Early approaches to OBIA performed image segmentation by the classical split and merge approach. Here, a meaningful partition was defined by applying a split process to define a set of elementary (homogeneous) regions that are then merged under certain conditions [1]. The latter may be based on geometric attributes like area, texture attributes like statistical moments of intensity distribution, shape attributes like shape factors, or any of their combinations [2–4]. On the other hand, more recent approaches to OBIA focus on hierarchical image segmentations that are based on scale-space representation, i.e. a set of image segmentations at different detail levels, in which the segmentation at finer levels are nested with respect to those at coarser levels [1, 5]. Some popular examples of such hierarchies include max-tree [6], α-tree [7, 8], and watershed hierarchies [9]. Unfortunately, hierarchical segmentation results in a huge number of nested partitions, which have to be merged efficiently. The region merging becomes in this way one of the most critical parts of the segmentation process.

Region merging can be considered from different theoretical aspects. A set merging problem, which has a long history in computing, is the first of them. Hopcroft and Ullman [10] proposed two algorithms based on quadtrees, both working in $O(n \log n)$ time, where n is the number of elements in the sets. In the first algorithm, the elements can be placed only in the leaves, while, in the second algorithm, the elements can exist in any of the tree vertices. Another tree-based algorithm was proposed by Tarjan [11] with the time complexity of $O(m \ \alpha(m, n))$, where $\alpha(m, n)$ is related with the inverse Ackermann function, while m and n correspond to the numbers of elements in both sets. His algorithm was also used by Najman et al. [9] for hierarchical watershed cuts. Tarjan and van Leeuwen [12] performed the worst-case analysis of the algorithms and concluded that the linear-time set-merging algorithm remains an open problem. Cormen et al. [13] also considered merging of the disjoint sets using either linked lists or trees. Another solution for merging regions was introduced by Horowitz and Pavlidis [14]. This method is also based on quadtrees, with, as pointed out by Brun and Domenger, considerable limitations [15]. They recognised that the regions differ importantly from the classical understanding of the sets. Namely, the elements of the regions also have spatial attributes (i.e. raster coordinates), and, therefore, it is possible to determine the border of the regions uniquely. Brun and Domenger developed a method by placing the image in the Khalimsky plane [16]. The region is considered as a set of topological maps which are mapped in the Euclidean plane. Another approach is based on the theory of geometric and solid modelling [17, 18], where merging is considered as a special case of the Boolean union. The so-called regularised Boolean operations were introduced to preserve the dimension homogeneity of the resulting object [19]. The solution is, typically, found in two steps. First, the intersection points between the involved geometric objects are determined, and second, the resulting shape is determined by the so-called walkabout. In 2D, the first part is solved in the expected time $O((n + m) \log(n + m) + I)$, where n and m are the number of vertices determining the input polygons, and I is the number of actual intersections [20]. If the proper data structure is used, the second step is realised in linear time. Such data structures have been proposed by Grainer and Horman [21], Vatti [22], and Liu et al. [23]. Rivero and Feito [24] proposed an approach for Boolean operations on polygons based on the theory of simplices. Their idea was later improved in ref. [25]. Very recently, an algorithm for Boolean operations for rasterised shapes was presented in ref. [26]. A space-filling curve was applied for the determination of the intersected pixels, while the walkabout was performed with a Greiner and Horman-like data structure. The proposed geometric approaches, however, cannot be applied in the OBIA, as they are based on the theory of regularised Boolean operations, which preserves the dimensional homogeneity of the resulting objects strictly. Consequently, this approach cannot handle all possible cases which may appear during region growth.

In this chapter, a new solution is proposed for a general region merging problem suitable for hierarchical OBIA. The main contributions are a theoretical derivation of the merging function in the raster space, represented by a 4-connected graph, and a proposal of an efficient implementation based on chain codes that ensure compact region representation.

The chapter is structured in five sections. Section 2 introduces the problem and formalises it. Brief implementation hints are given in Section 3. Section 4 presents empirical results, while Section 5 concludes the chapter.

2. Definitions

The key terms, needed to present the problem and to derive its formal solution, are defined in this section. Among other concepts, the region, raster space, and region merging are defined, which appeared in the title of this chapter.

Directed graph. $G = (V, E)$ defined by a vertex set $V = \{v_i\}$ and an edge set $E = \{e_{i,j}\}$ is a directed graph if E is given by ordered pairs (directed edges) of vertices $e_{i,j} = (v_i, v_j)$.

Raster space. Let $G = (V, E)$ be a directed graph. If V is determined by regularly spaced vertices $v_i = (x_i, y_i)$ with integer coordinates $x_i \in [0, X]$ and $y_i \in [0, Y]$, and E imposes 4-connectivity on them, then G defines the raster space. In other words, for each pair of adjacent vertices v_i, v_j linked by edge $e_{i,j} \in E$, there exists either relation $e_{i,j} \rightarrow \left(x_j, y_j\right) = (x_i \pm 1, y_i)$ or $e_{i,j} \rightarrow \left(x_j, y_j\right) = (x_i, y_i \pm 1)$. Each vertex can also be linked to itself, thus, $\forall v_i \in V \rightarrow e_{i,i} \in E$.

Intuitively, a region is a group of connected raster cells (grid cells or pixels). It may be represented either as a collection of the pixels themselves or by its boundary. This second possibility is used in this work. It is based on the concepts of trail and cycle which must, therefore, be introduced first.

Trail. Trail $t_{i_0,i_L} = \langle v_{i_0}, v_{i_1}, \ldots, v_{i_L} \rangle$ in G with length L is a sequence of adjacent vertices where for each pair $v_{i_l}, v_{i_{l+1}} \in t_{i_0,i_L} \rightarrow e_{i_l,i_{l+1}} \in E$. Trails t_{i_0,i_L} and t_{j_0,j_K} are connected if they share at least one subtrail, i.e. if a set of subtrails $T = t_{i_0,i_L} \cap t_{j_0,j_K} \neq \emptyset$.

Figure 1 shows two cases of connected trails. Trails $t_{0,6}$ and $t_{7,8}$ are connected through subtrail $t_{4,4} = \langle v_4, v_4 \rangle$ in **Figure 1a**, while trails $t_{0,6}$ and $t_{9,7}$ in **Figure 1b** share two subtrails, namely $T = t_{0,6} \cap t_{9,7} = \{t_{1,1}, t_{3,5}\}$, where $t_{1,1} = \langle v_1, v_1 \rangle$ and $t_{3,5} = \langle v_3, v_4, v_5 \rangle$. The shared subtrails will be hereinafter referred to as the intersection trails.

Trail t_{i_0,i_L} can be split into two trails t_{i_0,i_l} and t_{i_{l+1},i_L} at any $v_{i_l} \in t_{i_0,i_L}$. t_{i_0,i_L} is, therefore, a concatenation of t_{i_0,i_l} and t_{i_{l+1},i_L} as formally shown in Eq. (1):

$$t_{i_0,i_L} = t_{i_0,i_l} {}^\frown t_{i_{l+1},i_L}. \tag{1}$$

Cycle. Trail $t_{i_0,i_L} = \langle v_{i_0}, v_{i_1}, \ldots, v_{i_{L+1}} \rangle$ is cycle $c_{i_0,i_L} = \langle v_{i_0}, v_{i_1}, \ldots, v_{i_L} \rangle$, if $i_0 = i_{L+1}$. As each vertex can be linked to itself, the smallest cycle $c_{i_0,i_0} = \langle v_{i_0} \rangle$ is defined by trail $t_{i_0,i_0} = \langle v_{i_0}, v_{i_0} \rangle$. Contrary to the traditional definition of cycle, we do require that all vertices, except the end vertices, are distinct in c_{i_0,i_L}. Any cycle can, because of this, be composed of more than one cycle, where intermediate vertices are contained more than once.

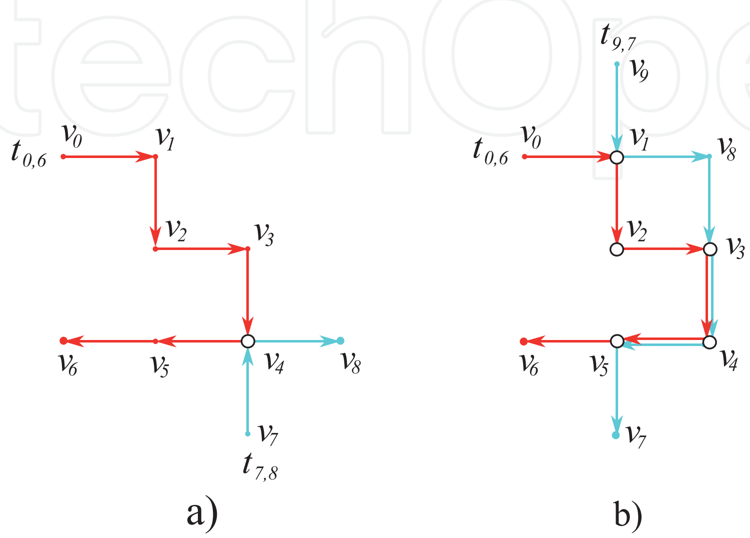

Figure 1.
Connected trails: (a) $t_{0,6} = \langle v_0, v_1, v_2, v_3, v_4, v_5, v_6 \rangle$, $t_{7,8} = \langle v_7, v_4, v_8 \rangle$, *(b)* $t_{0,6} = \langle v_0, v_1, v_2, v_3, v_4, v_5, v_6 \rangle$, $t_{9,7} = \langle v_9, v_1, v_8, v_3, v_4, v_5, v_7 \rangle$.

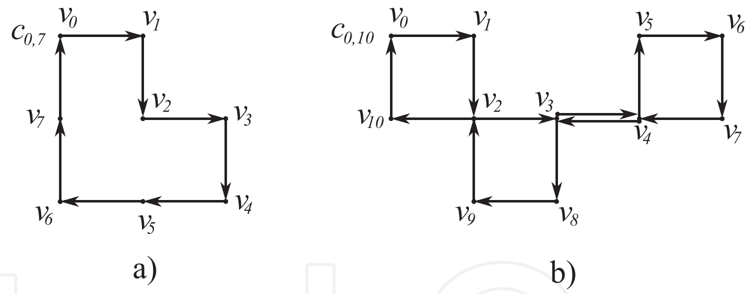

Figure 2.
Cycles: (a) $c_{0,7} = \langle v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7 \rangle$, *(b)* $c_{0,10} = \langle v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_4, v_3, v_8, v_9, v_2, v_{10} \rangle$.

Figure 2 shows examples of two cycles. The one in **Figure 2a** contains each vertex exactly once, while vertices v_2, v_3, and v_4 are contained twice in the cycle in **Figure 2b**. Note that any cycle can be rotated by any number of vertices $0 < l \le L$, i.e. $c_{i_0,i_L} = \langle v_{i_0}, v_{i_1}, \dots, v_{i_L} \rangle = \langle v_{i_l}, v_{i_{l+1}}, \dots, v_{i_L}, v_{i_0}, v_{i_1}, \dots, v_{i_{l-1}} \rangle = c_{i_l,i_{l-1}}$. Its decomposition can then be described as concatenation from Eq. (2):

$$c_{i_0,i_L} = t_{i_l,i_k} {}^\frown t_{i_{k+1},l_{L-1}}. \qquad (2)$$

As shown in **Figure 3**, any subtrail $t_{i_l,i_k} \subseteq c_{i_0,i_L}$, $0 \le l, k \le L$, can be removed from cycle c_{i_0,i_L} according to Eq.(3). The obtained result is also a subtrail.

$$t_{i_{k+1},i_{l-1}} = c_{i_0,i_L} \backslash t_{i_l,i_k}. \qquad (3)$$

Let $c_{i_0,i_3} = \langle v_{i_0}, v_{i_1}, v_{i_2}, v_{i_3} \rangle$ be an elementary clockwise oriented cycle, where $v_{i_0} = (x_i, y_i), v_{i_1} = (x_i, y_i + 1), v_{i_2} = (x_i + 1, y_i + 1)$, and $v_{i_3} = (x_i + 1, y_i)$. This elementary cycle defines a grid cell, with its interior on the right side of each edge $e_{i_l,i_{l+1}} = (v_{i_l}, v_{i_{l+1}})$, $0 \le l \le 3$ as shown in **Figure 4**.

Region. The region R is either a grid cell defined by an elementary clockwise oriented cycle or a group of grid cells bounded by the resulting cycle(s) of the region merging function (defined soon after this definition). For simplicity, a region will be equated with its boundary in the continuation, i.e. R will be treated as a cycle or a set of cycles.

Figure 5 shows the result of merging two elementary cycles c_{i_0,i_L} and c_{j_0,j_K} $(L = K = 3)$, which share either an edge (**Figure 5a**) or a vertex (**Figure 5b**). It indicates that the resulting merged region is defined by a concatenation of both

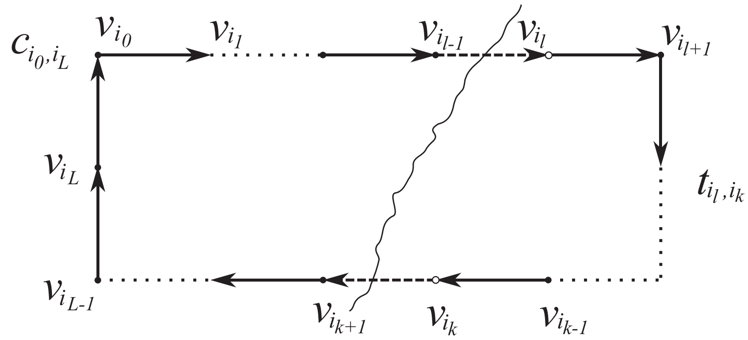

Figure 3.
Removing trail t_{i_l,i_k} *(on the right) from cycle* c_{i_0,i_L} *results in subtrail* $t_{i_{k+1},i_{l-1}}$ *(on the left).*

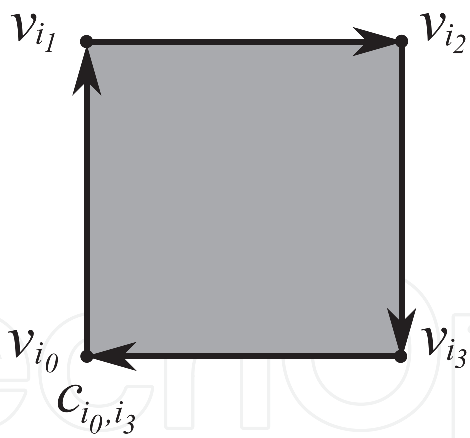

Figure 4.
Elementary cycle c_{i_0,i_3} enclosing a grid cell.

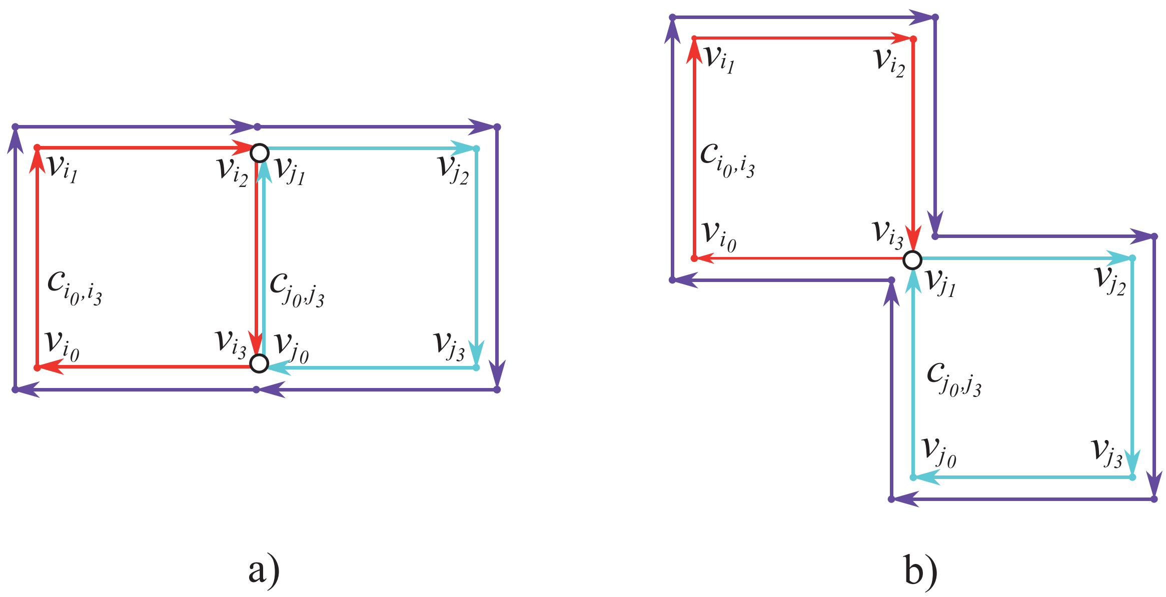

Figure 5.
Merging two elementary cycles with a shared edge (a) and vertex (b); the resulting cycles are in violet.

elementary cycles without their intersection $c_{i_0,i_L} \cap c_{j_0,j_K}$. Note, however, that c_{i_0,i_L} and c_{j_0,j_K} are both oriented in the clockwise direction, and, therefore, the orientation of the shared edges is opposite. To make them equal and, thus, to make their intersection non-empty, the orientation changing operation $\leftarrow$ is defined here, such that $\overleftarrow{c_{i_0,i_L}} = \langle v_{i_L}, v_{i_{L-1}}, \dots, v_{i_0} \rangle$. **Figure 6** shows an example of merging two non-elementary cycles which still share a single, but longer intersection trail. A similar conclusion as above may be made. The region merging function may be formally defined now.

Region merging function. Two cycles c_{i_0,i_L} and c_{j_0,j_K}, which share a single intersection trail t_{i_l,i_k} can be merged into a region R by a merging function $\mathcal{M}$ defined by Eq. (4):

$$\begin{aligned}
\mathcal{M}\left(c_{i_0,i_L}, c_{j_0,j_K}, t_{i_l,i_k}\right) &= \left(c_{i_0,i_L} \backslash \left(c_{i_0,i_L} \cap \overleftarrow{c_{j_0,j_K}}\right)\right)^\frown \langle v_{i_l} \rangle^\frown \left(c_{j_0,j_K} \backslash \left(c_{j_0,j_K} \cap \overleftarrow{c_{i_0,i_L}}\right)\right)^\frown \langle v_{i_k} \rangle = \\
&= \left(c_{i_0,i_L} \backslash t_{i_l,i_k}\right)^\frown \langle v_{i_l} \rangle^\frown \left(c_{j_0,j_K} \backslash t_{j_n,j_m}\right)^\frown \langle v_{i_k} \rangle = \\
&= t_{i_{k+1},i_{l-1}}{}^\frown \langle v_{i_l} \rangle^\frown t_{j_{m+1},j_{n-1}}{}^\frown \langle v_{i_k} \rangle = \\
&= t_{i_{k+1},i_{l-1}}{}^\frown \langle v_{j_m} \rangle^\frown t_{j_{m+1},j_{n-1}}{}^\frown \langle v_{j_n} \rangle.
\end{aligned} \tag{4}$$

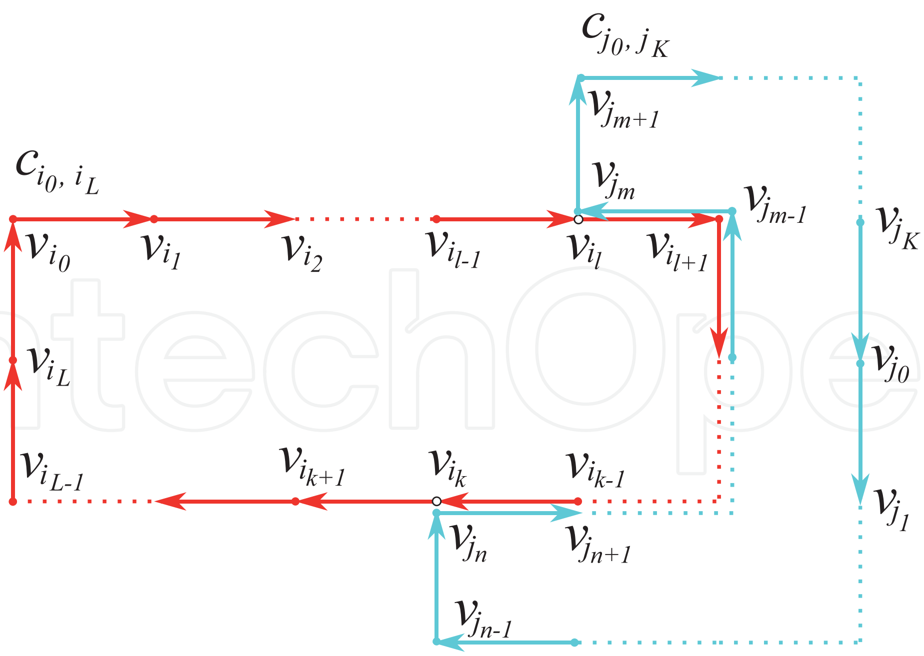

Figure 6.
Merging of two non-elementary cycles whose intersection trail is a longer sequence of edges.

In a general case, where the intersection of the input cycles consists of more trails, the merged region R is defined by Eq. (5):

$$R = \bigcap_{t_{i_l,i_k}^{(h)} \in T_i} \mathcal{M}\left(c_{i_0,i_L}, c_{j_0,j_K}.t_{i_l,i_k}^{(h)}\right). \tag{5}$$

Eq. (4) is thus applied when $|T_i| = |T_j| = 1$, where $T_i = \{t_{i_k,i_l}\} = c_{i_0,i_L} \cap \overleftarrow{c_{j_0,j_K}}$ and $T_j = \{t_{j_m,j_n}\} = c_{j_0,j_K} \cap \overleftarrow{c_{i_0,i_L}}$. On the other hand $|T_i| = |T_j| > 1$ implies utilisation of

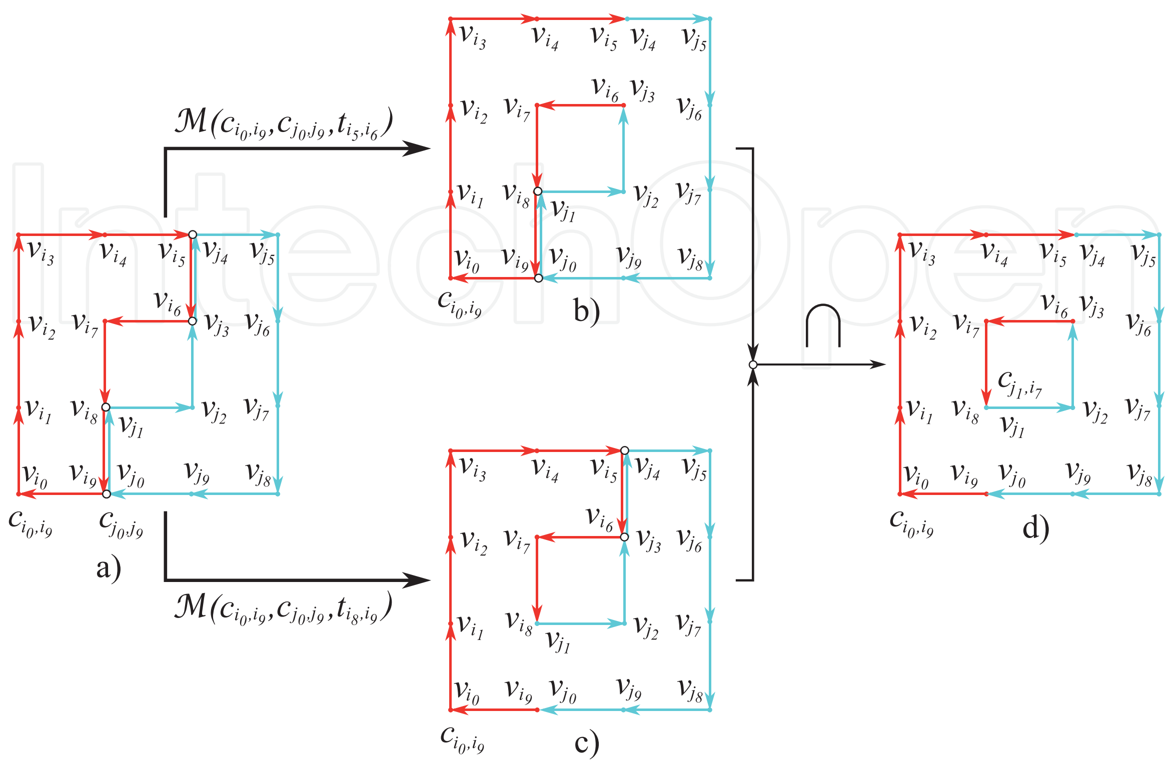

Figure 7.
Applying Eq. (5) to merge two cycles whose intersection consists of two intersection trails, i.e. $|T_i| = 2$.

Eq. (5) and each intersection trail then results in a new cycle. $|T_i|$ cycles are, therefore, constructed, and the resulting region is described by the intersection of these cycles. Note that h, such that $0 \le h < |T_i|$, is an index of the intersection trail in Eq. (5). It is obvious that Eq. (5) is valid also when $|T_i| = 1$.

Figure 7 shows an illustrative example. The set of intersection trails is $T_i = \{t_{i_5,i_6}, t_{i_8,i_9}\}$. Applying Eq. (4) on the intersection trail $t_{i_5,i_6} \in T_i$ results in **Figure 7b**. Similarly, using Eq. (4) on the second intersection trail $t_{i_8,i_9} \in T_i$ gives **Figure 7c**. The final result is then obtained as an intersection (Eq. (5)) between cycles from **Figure 7b** and **c**. As seen in **Figure 7d**, the resulting region R consists of two cycles, which are exactly the same as $|T_i|$.

3. Implementation

The concept of chain codes is used to represent regions $R \subseteq G$ in the presented method. The chain code, introduced by Freeman [27], consists of a few simple commands by which navigation through the edges of G is made possible. Freeman proposed two chain codes known as Freeman chain code in eight ($F8$) and four ($F4$) directions. Other chain codes were discovered by Bribiesca (Vertex Chain Code, *VCC*) [28], Sánchez-Cruz and Rodríguez-Dagnino (Three-Orthogonal chain code, 3*OT*) [29], Žalik et al. (Unsigned Manhattan Chain Code, *UMCC*) [30], and Dunkelberger and Mitchell (Mid-crack Chain Code) [31]. In general, there are two types of chain codes: those operating on raster pixels and those working with raster edges. The latter is known as crack-chain codes [32, 33]. $F4$ is the only chain code which can be used in both contexts, and its crack interpretation is used in this algorithm.

The $F4$ alphabet consists of four commands/symbols $\sigma_i \in \Sigma_{F4}$, $\Sigma_{F4} = \{0, 1, 2, 3\}$, shown in **Figure 8**. Let $\langle \sigma_i \rangle$ be the sequence of $F4$ commands. To embed the chain code in G, the position of the chain code starting vertex v_0 is needed, while the positions of the remaining vertices are determined from the $F4$ commands according to Eq. (6):

$$v_{i+1} = \begin{cases} (x_i + 1, y_i), & \text{if } \sigma_i = 0; \\ (x_i, y_i - 1), & \text{if } \sigma_i = 1; \\ (x_i - 1, y_i), & \text{if } \sigma_i = 2; \\ (x_i, y_i + 1), & \text{if } \sigma_i = 3. \end{cases} \tag{6}$$

Figure 8b shows the elementary cycle c_{i_0,i_3} determined as $v_{i_0}\langle 1, 0, 3, 2\rangle$, where $v_{i_0} = \left(x_{i_0}, y_{i_0}\right)$ are the coordinates of the cycle's starting vertex.

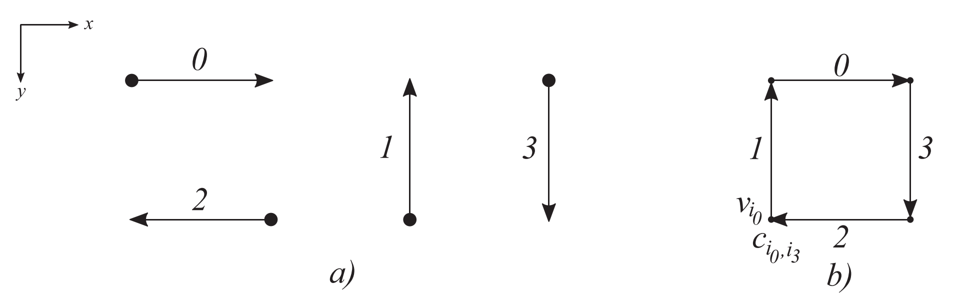

Figure 8.
F4 chain code symbols (a); the elementary cycle described by F4 chain code symbols (b).

Any region in G can be represented in this way. For example, regions containing more cycles are shown in **Figure 9**, where, for better presentation, the inner cells of the region are shadowed. According to the definitions from Section 2, a vertex $v_i \in R$ can be part of two cycles at the same time, or, within each cycle, v_i can be passed twice, too. In the continuation, the cycle corresponding to the outer border of R is considered as a *loop*, while cycles representing holes are named *rings* [19]. The orientation of the loop is clockwise, while the rings' is the opposite (**Figure 9**).

A data structure for representing R is shown schematically in **Figure 10**. It consists of an array of starting points and an array of $F4$ chain code sequences. The loop is always located at index 0, and k, $k \geq 0$, rings follow in an arbitrary order. The algorithm, which implements Eq. (5), consists of two main steps:

- Determining the intersection trails T_i between cycles of input regions and

- Realising Eq. (5) by performing a walkabout through the edges not in T_i.

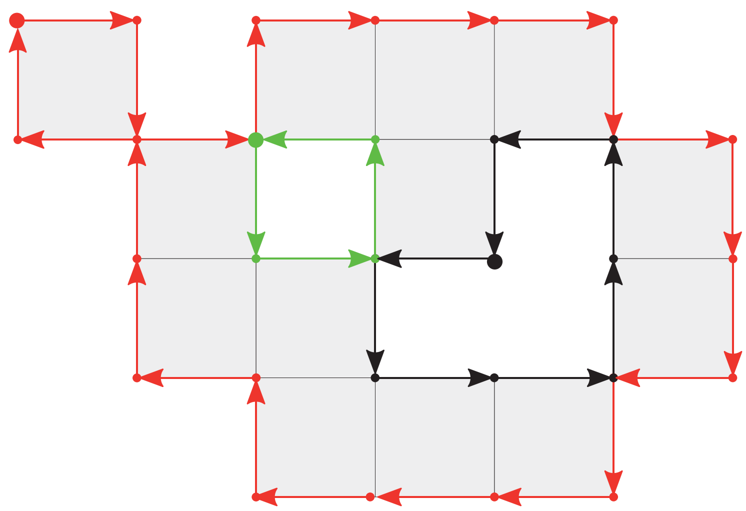

Figure 9.
Region with two rings: The rings' vertices (green, black) can be shared with the loop (red); the vertices within the loop can be used twice.

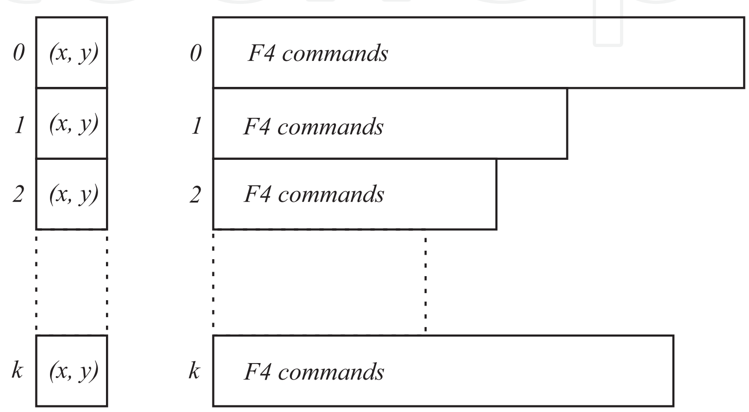

Figure 10.
Data structure of region R: Array of starting points (left) of individual cycles and the corresponding sequences of F4 chain codes (right).

3.1 Determining the intersection trails

Determination of intersection points between edges of regions can be related with the problem of finding intersections between polygon edges. As the naive implementation of the latter works with $O(n^2)$ time complexity, various approaches were suggested to reduce it [13, 34–36]. The presented solution exploits the fact that regions R_i and R_j are embedded into common directed graph G, i.e. $R_i \in G \wedge R_j \in G$. The following data are associated with each vertex $v_i \in G$:

- two pointers (P_{i1} and P_{i2}) into an array of $F4$ symbols for region R_i (one or both pointers can be NULL),
- two pointers (LR_{i1} and LR_{i2}) pointing to the loop, or to the corresponding ring of region R_i (similarly as above, one or both pointers can be NULL), and
- the same information for region R_j.

Let us consider the example in **Figure 11**, where the loop's edges of R_i are plotted in red, the edges of its ring in black, while edges of region R_j are in cyan. The content of data structures for both regions is given in **Table 1**. **Table 2** shows the information of some characteristic vertices in G. Vertex v_a, for example, belongs to R_i, its P_{i1} points to the index 1 in the array of $F4$ chain codes, and the vertex belongs to the loop ($LR_{i2} = 0$). Vertex v_f is met two times by the edges of the R_i loop and, therefore, two pointers are pointing to the 3rd and 15th positions. Vertex v_h is the most interesting. In this vertex three cycles are met. Pointer P_{i1} points to the 7^{th} R_i loop

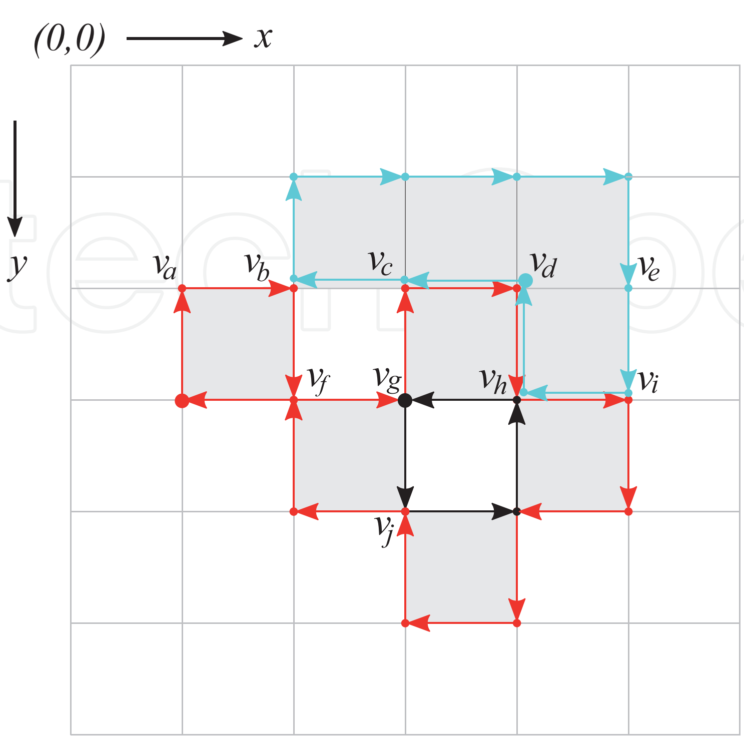

Figure 11.
Regions R_i (red and black) and R_j (cyan) embedded into G, and some characteristic vertices considered in ***Table 2***.

			i:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
R_i	0	(1, 3)	$F4$:	1	0	3	0	1	0	3	0	3	2	3	2	1	2	1	2
			i:	0	1	2	3												
	1	(3, 3)	$F4$:	3	0	1	2												
R_j			i:	0	1	2	3	4	5	6	7	8	9						
	0	(4, 2)	$F4$:	2	2	1	0	0	0	3	3	2	1						

Table 1.
*Data structures for regions R_i and R_j from **Figure 11**.*

Vertex in Figure 11	P_{i1}	P_{i2}	LR_{i1}	LR_{i2}	P_{j1}	P_{j2}	LR_{j1}	LR_{j2}
v_a	1	/	0	/	/	/	/	/
v_b	2	/	0	/	2	/	0	/
v_c	5	/	0	/	1	/	0	/
v_d	6	/	0	/	0	/	0	/
v_e	/	/	/	/	7	/	/	/
v_f	3	15	0	0	/	/	/	/
v_g	4	0	0	1	/	/	/	/
v_h	7	3	0	1	9	/	0	/
v_i	8	/	0	/	8	/	0	/
v_j	13	1	0	1	/	/	/	/
(NULL pointers are marked with '/')								

Table 2.
*The content of G at specific vertices marked in **Figure 11**.*

position, and P_{i2} points to the 3^{rd} position of the first R_i ring. The loop of R_j is accessed by the chain code command stored at position 9 in the $F4$ array.

Having marked the vertices in G properly, it is easy to determine the intersection trails. The region with the smallest number of edges is found (let us suppose it is R_j), and all its vertices are visited. The sequence of edges marked with pointers of both regions is identified as being a part of the intersection trail.

3.2 Performing the walkabout

Those trails which were not labelled as the intersection ones, are united into the new region by the algorithm, consisting of the following steps:

1. Mark all edges from intersection trails as *visited* and the remaining edges as *not visited*.

2. Find an arbitrary non-visited edge $e \in R_j$. If such edge does not exist, jump to step 9.

3. The initial vertex of e is chosen as the starting vertex v_s for the walkabout. An empty queue Q is created.

4. Walk along the edges of R_j, mark each passed edge as *visited* and store passed $F4$ commands into Q until v_s is met, or the vertex with set R_i pointer is reached.

5. If v_s is reached, go to step 8, otherwise switch to the edge of region R_i using the pointer stored at the vertex from G.

6. Walk along the edges of R_i, mark each passed edge as *visited* and store passed $F4$ commands into Q until the v_s is met or the vertex with R_j pointer is detected.

7. If v_s is not reached yet, switch to the region R_j using the pointer stored at the considered vertex, and go to step 4, otherwise go to the next step.

8. Store the obtained cycle into the list of cycles LoC and return to step 2.

9. Insert non-visited cycles of input both regions R_i and R_j into LoC.

10. Find the loop in LoC; all others cycles in LoC represent rings of the merged region R.

These steps are highlighted in Algorithm 1. The decision whether the cycle defines a loop or a ring depends on its orientation. It can be determined by Eq. (7), where $Q = \langle \sigma_i \rangle$ ($F4$ chain code commands σ_i are treated as integers for this purpose).

$$o = \sum_{i=0}^{|Q|-1} \begin{cases} -1 & : \sigma_i = 0 \wedge \sigma_{(i+1) mod \ |\Sigma_{F4}|} = |\Sigma_{F4}| - 1 \\ 1 & : \sigma_i = |\Sigma_{F4}| - 1 \wedge \sigma_{(i+1) mod \ |\Sigma_{F4}|} = 0 \\ \sigma_{(i+1) mod \ |\Sigma_{F4}|} - \sigma_i & : \text{otherwise} \end{cases} \tag{7}$$

The equation evaluates each right turn with -1 and each left turn with 1. The clockwise oriented cycles result in $o = -4$, while the counter-clockwise cycles achieve $o = 4$.

Algorithm 1 Merging regions R_i and R_j.

function MERGE(G, R_i, R_j)
  ▹ function merges regions R_i and R_j embedded in graph G
  Insert(G, R_i, *NonVisited*) ▹ insert region into G, mark edges as *NonVisited*
  Insert(G, R_j, *NonVisited*)
  MarkIntersectionEdges(G, R_i, R_j) ▹ intersection edges are marked as *Visited*
  e = GetNonVisitedEdge(R_j)
  $LoC = \{ \}$ ▹ List of Cycles should be empty
  while Visited(R_j, e) = *NonVisited* **do**
    CycleFound = **false**
    $Q = \{ \}$ ▹ clear the queue containing F4 symbols
    Q =AddF4Symbol(Q, R_j, e) ▹ add F4 chain code symbol of e
    v_s = ReturnVertex(R_j, e) ▹ store the starting vertex

MarkVisitedEdge(R_j, e) ▷ mark edge as visited
e = NextEdge(R_j, e) ▷ move one edge forward along R_j boundary
repeat
while (Visited(R_j, e) = *NonVisited*) AND
(ReturnVertex$(R_j, e) \neq v_s$) **do** ▷ walk along edges of R_j
Q = AddF4Symbol(Q, R_j, e) ▷ add F4 chain code symbol of e
MarkVisitedEdge(R_j, e) ▷ mark edge as visited
e = NextEdge(R_j, e) ▷ move one edge forward along R_j boundary
end while
if ReturnVertex(R_j, e) = v_s **then** ▷ the cycle is formed
CycleFound = **true**
else ▷ otherwise continue the walk on R_i
e = ReturnEdgeFromOtherRegion(R_j, e) ▷ get $e \in R_i$
while (Visited(R_i, e) = *NonVisited*) AND ▷ walk along R_i edges
(ReturnVertex$(R_i, e) \neq v_s$) **do**
Q = AddF4Symbol(Q, R_i, e)
MarkVisitedEdge(R_i, e)
e = NextEdge(R_i, e)
end while
if ReturnVertex(R_i, e) = v_s **then**
CycleFound = **true**
else ▷ jump to the R_j boundary again
e = ReturnEdgeFromOtherRegion(R_i, e) ▷ get $e \in R_j$
end if
end if
until *CycleFound* = **true**
LoC = StoreCycle(v_s, Q) ▷ store constructed cycle
e = GetNonVisitedEdge(R_j) ▷ get next non-visited edge
end while
AddNonVisitedCycles(LoC, R_i) ▷ add non-visited cycles (holes)
AddNonVisitedCycles(LoC, R_j)
DetermineLoop(LoC) ▷ among all cycles the loop is found by (Eq. 7)
R = ConstructRegion(LoC)
return R
end function.

4. Analysis of the algorithm

4.1 Time and space complexity estimation

The proposed algorithm consists of three parts: finding the intersection trails, performing the walkabout, and determination of loop and rings.

Let the four-connected graph G consist of n vertices. Let k_i and k_j represent the number of $F4$ edges defining cycles of R_i and R_j, respectively. $k_i = k_j = k$ may be assumed without loss of generality. At first, both regions are embedded into G. This is done in $T_e(k) = T(k_i) + T(k_j) = 2k$ time. One of the regions is walked-about and the edges of the intersection trails are determined in time $T_w(k) = k$. The first part of the algorithm is, therefore, executed in $T_1(k) = T_e(k) + T_w(k) = 3k$ time.

During the walkabout, all edges of both regions not being part of the intersection trail, are visited exactly once. In the worst case, all $2k$ edges must be visited in time $T_2(k) = 2k$.

The orientation of the cycles of the merged region is determined in the last step. This task is also terminated in $T_3 = 2k$ time, as the merged region cannot have more than $2k$ edges.

The proposed merging algorithm is, therefore, realised in $T(k) = T_1(k) + T_2(k) + T_3(k) = 3k + 2k + 2k = 7k = O(k)$. k cannot be greater than n, and therefore, one merging operation is terminated in the worst case in $O(n)$ time. However, in the majority of cases, $k < < n$. In such, an expected case, one merging operation is terminated in a constant time $O(1)$.

The algorithm needs memory for G with n vertices, i.e. $S_G(n) = n$. In addition, two regions need to be stored. In the worst case, both regions require additional $S_R(n) = n$ memory space. The algorithm, therefore, works in $S(n) = 2n = O(n)$ space.

4.2 Experiment

The standard benchmark images shown in **Figure 12** have been used in the experiment with different resolutions. A criterion for merging two neighbouring regions was the colour similarity. By doubling the size of the image in both directions iteratively, the number of pixels n is increasing by the power of 4, and the number of required merging operations follows this exponential growth; actually, the number of merging operations is $n - 1$.

Table 3 shows spent CPU time while performing merging from single pixels up to the entire image. A personal computer with a 3,5 GH Intel® Core™ i5–6600 K processor and 32 G bytes of RAM was used in the experiment. The program was implemented in C++ and compiled with C++ Visual Studio 2019 under the Windows 10 operating system. As can be seen, the actual CPU time spent depends on the colour characteristics of the images. The image Lenna has large parts of very similar colours and therefore, the regions grow rapidly which is reflected in the shortest CPU spent time. The image Peppers exposes a similar characteristic. On the other hand, the image Baboon consists of very small homogeneous regions, reflecting in longer CPU time spent.

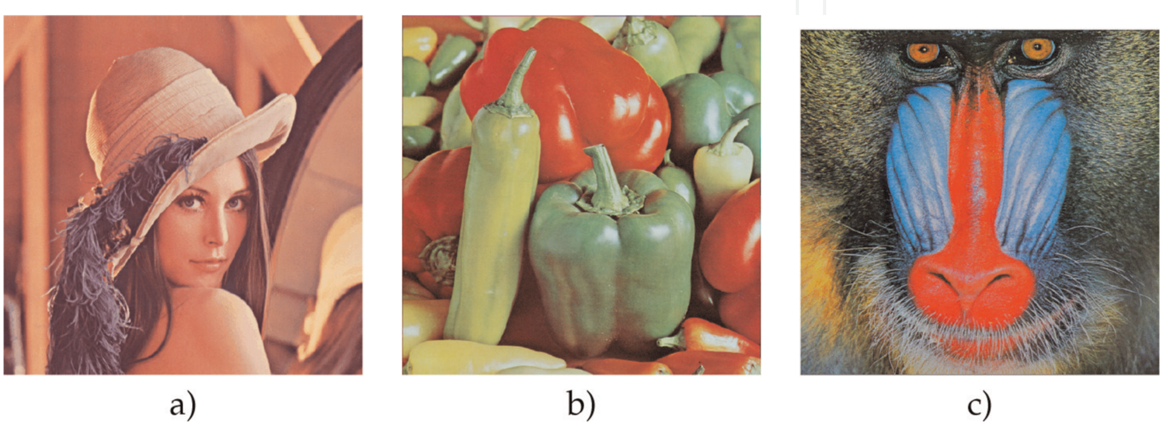

Figure 12.
Images used in the experiments: (a) Lenna, (b) peppers, and (c) baboon.

Size (pixels)	n	t_L (s)	$t_P(s)$	Size (pixels)	n	$t_B(s)$
64 × 64	4096	0.014	0.017	63 × 60	3780	0.018
128 × 128	16,384	0.081	0.064	125 × 120	15,000	0.169
256 × 256	65,536	0.715	0.624	250 × 240	60,000	1.682
512 × 512	262,144	7.885	11.778	500 × 480	240,000	20.454

Table 3.
Spent CPU time for images Lenna (L), peppers (P), and baboon (B) at different resolutions.

Size (pixels)	t_L (s)	t_P (s)	Size (pixels)	t_B (s)
64 × 64	0.683	0.602	63 × 60	0.892
128 × 128	8.765	8.646	125 × 120	13.823
256 × 256	219.450	211.762	250 × 240	247.323
512 × 512	OOM[a]	OOM[a]	500 × 480	OOM[a]

[a]*OOM denotes out-of-memory.*

Table 4.
Spent CPU time with the referenced approach.

We implemented the set-based version of the merging operation for comparison. In this case, the region is represented by the STD structure *unordered_map*. The obtained results are shown in **Table 4**. As can be seen, the set-based approach performs considerably slower. In addition, it obviously spends more memory, as images with the highest resolutions cannot be processed any more.

5. Conclusion

A new region merging algorithm, suitable for hierarchical object-based image analysis, is proposed in this chapter. The raster space is represented by a 4-connected graph, and a merging function is derived formally upon it. The implementation follows the theoretical investigation strictly. The edges forming the border of the region embedded in the 4-connected graph are represented by the Freeman crack chain code in four directions. The implementation works in two main steps: a determination of the common vertices and edges of the regions being merged, and a walkabout, which realises the theoretically derived merging function. A classification of the obtained region's edges to those representing the holes and those defining the outer border, may be done at the end.

The algorithm's worst-case time complexity is $O(n)$ for one merging operation, where n is the number of graph vertices. However, as the number of edges defining the two regions being merged is much smaller than n, the expected time complexity is actually independent on n, i.e. the expected time complexity of the proposed algorithm is $O(1)$.

In addition to the methodical implementation of the region merging procedure, the proposed chain code-based approach enables efficient extraction of various essential shape descriptors [3]. The approaches for extracting these descriptors can be divided roughly into the region- and contour-based approaches, and the latter are known as being computationally demanding for traditional hierarchical segmentation and region growing. Namely, they require the boundary to be extracted after each region merging operation. Because of this, these are rarely used, e.g. as stopping criteria during the region growing, or as thresholds for hierarchical cuts. On the other hand, chain codes by themselves allow for efficient description of shapes.

Acknowledgements

The authors acknowledge the financial support from the Slovenian Research Agency (Research Core Funding No. P2-0041 and Project No. N2-0181), and the Company IGEA, d.o.o., for co-financing this research.

Thanks

Thanks to Anže Ferčec, who implemented the referenced merging algorithm.

Nomenclature

c	cycle
CPU	central processing unit
e	edge
E	set of edges
$F4$	Freeman chain code in four directions
G	directed graph
L	the number of vertices in a trail or in a cycle
LR	pointer to the loop/ring of the region
LoC	list of cycles
$\mathcal{M}$	merging function
OBIA	Object-Based Image Analysis
P	pointer to the region
Q	queue
R	region
Σ_{F4}	alphabet of Freeman's chain code in four directions
σ	$F4$ symbol
t	trail
T	set of trails
v	vertex
V	set of vertices

Author details

Borut Žalik*†, David Podgorelec†, Niko Lukač†, Krista Rizman Žalik†
and Domen Mongus†
Faculty of Electrical Engineering and Computer Science, University of Maribor,
Maribor, Slovenia

*Address all correspondence to: borut.zalik@um.si

† These authors contributed equally.

References

[1] Guimarães SJF, Cousty J, Kenmochi Y, Najman L. A Hierarchical Image segmentation algorithm based on an observation scale. In: Gimel'farb G, Hancock E, Imiya A, Kuijper A, Kudo M, Omachi S, Windeatt T, Yamada K, editors. Structural, Syntactic, and Statistical Pattern Recognition. Lecture Notes in Computer Science;7626. Berlin, Heidelberg: Springer; 2012. pp. 116-125. DOI: 10.1007/978-3-642-34166-3_13

[2] Materka A, Strzelecki M. Texture analysis methods – a review. Lodz, Poland: COST B11 report, Institute of Electronics, Technical University of Lodz; 1998. Available from: https://www.researchgate.net/publication/249723259_Texture_Analysis_Methods_-_A_Review [Accessed: 2021-12-22]

[3] Zhang D, Lu G. Review of shape representation and description techniques. Pattern Recognition. 2004;**37**(1):1-19. DOI: 10.1016/j.patcog.2003.07.008

[4] Kurnianggoro L, Jo KH. A survey of 2D shape representation: Methods, evaluations, and future research directions. Neurocomputing. 2018;**300**:1-16. DOI: 10.1016/j.neucom.2018.02.093

[5] Xu Y, Carlinet E, Géraud T, Najman L. Efficient computation of attributes and saliency maps on tree-based image representations. In: Benediktsson J, Chanussot J, Najman L, Talbot H, editors. Mathematical Morphology and Its Applications to Signal and Image Processing. Lecture Notes in Computer Science; 9082. Berlin, Heidelberg: Springer; 2015. pp. 693-704. DOI: 10.1007/978-3-319-18720-4_58

[6] Salembier P, Oliveras A, Garrido L. Antiextensive connected operators for image and sequence processing. IEEE Transactions on Image Processing. 1998;**7**(4):555-570. DOI: 10.1109/83.663500

[7] Ouzounis GK, Soille P. Pattern spectra from partition pyramids and hierarchies. In: Soille P, Pesaresi M, Ouzounis GK, editors. Mathematical Morphology and Its Applications to Image and Signal Processing. Berlin, Heidelberg: Lecture Notes in Computer Science; 6671, Springer; 2011. pp. 108-119. DOI: 10.1007/978-3-642-21569-8_10

[8] Ouzounis GK, Soille P. The alpha-tree algorithm: Theory, algorithms and applications. Luxembourg: Technical reports JCR74511, European Commission, Joint Research Centre; 2012. DOI: 10.2788/48773

[9] Najman L, Cousty J, Perret B. Playing with Kruskal: Algorithms for morphological trees in edge-weighted graphs. In: Hendriks CLL, Borgefors G, Strand R, editors. Mathematical Morphology and Its Applications to Signal and Image Processing. Lecture Notes in Computer Science; 7883. Berlin, Heidelberg: Springer; 2013. pp. 135-146. DOI: 10.1007/978-3-642-38294-9_12

[10] Hopcroft JE, Ullman JD. Set merging algorithms. SIAM Journal of Computing. 1973;**2**(4):294-303. DOI: 10.1137/0202024

[11] Tarjan RE. Efficiency of a good but non-linear set union algorithm. Journal of ACM. 1975;**22**(2):215-225. DOI: 10.1145/321879.321884

[12] Tarjan RE, Leeuwen J. Worst-case analysis of set union algorithms. Journal of ACM. 1984;**31**(2):245-281. DOI: 10.1145/62.2160

[13] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. 3rd ed. Cambridge, London: MIT; 2009. p. 1292

[14] Horowitz SL, Pavlidis T. Picture segmentation by a tree traversal algorithm. Journal od ACM. 1976; **23**(2):368-388. DOI: 10.1145/321941.321956

[15] Brun L, Domenger JP. A new split and merge algorithm with topological maps. In: Proceedings of the 5th International Conference in Central Europe on Computer Graphics and Visualization (WSCG'97), 10–14 February 1996, Plzen, Czech Republic: University of West Bohemia. p. 21–31. Available from: https://www.researchgate.net/publication/2658919_A_New_Split_and_Merge_Algorithm_with_Topological_Maps [Accessed: 2021-12-22]

[16] Khalimsky E, Kopperman R, Meyer PR. Boundaries in digital planes. Journal of Applied Mathematics and Stochastic Analysis. 1990;**3**(1):27-55. DOI: 10.1155/S1048953390000041

[17] Hoffmann CM. Geometric and solid modeling: An introduction. San Francisco: Morgan Kaufmann; 1989. 338 p. Available from: https://dl.acm.org/doi/book/10.5555/74803 [Accessed: 2021-12-23]

[18] Mäntylä M. An introduction to solid modeling. New York: Computer Science Press; 1987. 401 p. Available from: https://dl.acm.org/doi/book/10.5555/39278 [Accessed: 2021-12-23]

[19] Mortenson ME. Geometric Modeling. 2nd ed. New York: Wiley; 1997. 523 p. Available from: https://dl.acm.org/doi/book/10.5555/248381 [Accessed: 2021-12-23]

[20] Mairson HG, Stolfi J. Reporting and counting intersections between two sets of line segments. In: Earnshaw R, editor. Theoretical Foundations of Computer Graphics and CAD. NATO ASI Series; F40. Berlin, Heidelberg: Springer; 1988. pp. 307-325. DOI: 10.1007/978-3-642-83539-1_11

[21] Greiner G, Hormann K. Efficient clipping of arbitrary polygons. ACM Transaction on Graphics. 1998;**17**(2):71-83. DOI: 10.1145/274363.274364

[22] Vatti BR. A generic solution to polygon clipping. Communications of ACM. 1992;**35**(7):56-63. DOI: 10.1145/129902.129906

[23] Liu YK, Wang XQ, Bao SZ, Gomboši M, Žalik B. An algorithm for polygon clipping, and for determining polygon intersections and unions. Computers & Geosciences. 2007;**33**(5):589-598. DOI: 10.1016/j.cageo.2006.08.008

[24] Rivero M, Feito FR. Boolean operations on general planar polygons. Computers & Graphics. 2000;**24**(6):881-896. DOI: 10.1016/S0097-8493(00)00090

[25] Peng Y, Yong JH, Dong WM, Zhang H, Sun JG. A new algorithm for Boolean operations on general polygons. Computers & Graphics. 2005;**29**(1):57-70. DOI: 10.1016/j.cag.2004.11.001

[26] Žalik B, Mongus D, Rizman Žalik K, Lukač N. Boolean operations on rasterized shapes represented by chain codes using space filling curves. Journal of Visual Communication and Image Representation 2017;49:420–432. DOI: 10.1016/j.jvcir.2017.10.003

[27] Freeman H. On the encoding of arbitrary geometric configurations. IRE Transactions on Electronic Computers. 1961;**EC10**(2):260-268. DOI: 10.1109/TEC.1961.5219197

[28] Bribiesca E. A new chain code. Pattern Recognition. 1999;**32**(2): 235-251. DOI: 10.1016/S0031-3203(98)00132-0

[29] Sánchez-Cruz H, Rodríguez-Dagnino RM. Compressing bi-level images by means of a 3-bit chain code. Optical Engineering. 2005;**44**(9): 097004. DOI: 10.1117/1.2052793

[30] Žalik B, Mongus D, Liu YK, Lukač N. Unsigned Manhattan chain code. Journal of Visual Communication and Image Representation 2016;**38**:186–194. DOI: 10.1016/j.jvcir.2016.03.001

[31] Dunkelberger KA, Mitchell OR. Contour tracing for precision measurements. St. Luis, USA: IEEE International conference on robotics and automation (ICRA); 25–28 March 1985. pp. 22-27. DOI: 10.1109/ROBOT.1985.1087356

[32] Wilson GR. Properties of contour codes. IEE Proceedins – Vision, Image and Signal Processing. 1997;**144**(3): 145-149. DOI: 10.1049/ip-vis:19971159

[33] Kabir S. A compressed representation of Mid-Crack code with Huffman code. International Journal on Image, Graphics and Signal Processing. 2015;7(10):11-18. DOI: 10.5815/ijigsp.2015.10.02

[34] de Berg M, Cheong O, van Kreveld M, Overmars M. Computational geometry: Algorithms and applications. 3rd ed. Berlin Heidelberg: Springer; 2008. p. 386. DOI: 10.1007/978-3-540-77974-2

[35] Chazelle B, Edelsbrunner H. An optimal algorithm for intersection line segments in the plane. Journal of ACM. 1992;**39**(1):1-54. DOI: 10.1145/147508.147511

[36] Žalik B. Two efficient algorithms for determining intersection points between simple polygons. Computers & Geosciences. 2000;**26**(2):137-151. DOI: 10.1016/S0098-3004(99)00071-0