CARPATHIAN J. MATH. Online version at http://carpathian.ubm.ro
**32** (2016), No. 1, 37 - 47 Print Edition: ISSN 1584 - 2851 Online Edition: ISSN 1843 - 4401

# On contour representation of two dimensional patterns

I. T. BANU-DEMERGIAN and G. STEFANESCU

ABSTRACT. Two-dimensional patterns are used in many research areas in computer science, ranging from image processing to specification and verification of complex software systems (via scenarios). The contribution of this paper is twofold. First, we present the basis of a new formal representation of two-dimensional patterns based on contours and their compositions. Then, we present efficient algorithms to verify correctness of the contour-representation. Finally, we briefly discuss possible applications, in particular using them as a basic instrument in developing software tools for handling two dimensional words.

## 1. INTRODUCTION

The study of two-dimensional shapes is of wide interest. Applicability in pattern recognition, image processing, computer graphics and, more recently, in interactive computation demonstrates the need of a compact and steady model to handle two-dimensional objects. Contours-based representations are often used as they fit efficiency and simplicity requirements. Among them, chain-codes allow compression, without losing any information. The general idea is to encode the border of an image by a list of line-segments characterized by length and direction. The first chain-code representation is due to Freeman, 1961 [7]. It describes a curve by linking adjacent points by one of eight possible moves, corresponding to an $i * 45^\circ$ angle, $i = 0..7$. Others encoding schemes, based on Freeman codification, have been proposed in [14, 13, 4]. Kaneko and Okudaira [13] obtained a high compression rate in coding geographic maps, using the property that a curve with gentle curvature is divided into long curve segments, each of which being represented by a sequence of two adjacent chain codes. E. Biribiesca [4] presented a formal language approach, specifying some algebraic properties of chain codes representing 3D curves.

We propose a chain code based on four directions, over rectangular grids. Each line segment in the boundary of a two-dimensional image is identified with a letter in the set $\{u, d, r, l\}$ ($u$ stands for "up", $d$ for "down", $r$ for "right", and $l$ for "left"), followed by a number, denoting its length. Roughly speaking, a *contour* is a closed line formed by a list of connecting segments, starting at a particular point, surrounding a finite internal area; a contour is associated to a bi-dimensional shape by a left-handed traversal. A general *2-dimensional word* is specified by filling the area delimited by a contour with letters form a given alphabet.

For (1-dimensional) words, a powerful representation is provided by finite automata, regular expressions, and Kleene algebras. The connection between Kleene algebras and finite automata [5] is well known and it provides a rich support for many others semantic models of computation, including models of parallel systems as: tile systems [9], Petri nets [8], timed automata [1], etc. A formalism for interactive parallel computation rv-IS (register-voice interactive systems) and a core programming language Agapia have

Received: 18.12.2013. In revised form: 10.12.2014. Accepted: 17.12.2014
2010 *Mathematics Subject Classification.* 68Q10, 68Q45, 68R01.
Key words and phrases. *Regular expressions, two-dimensional patterns, contours, structured interactive programming, formal methods.*
Corresponding author: I. T. Banu-Demergian; th_iulia84@yahoo.com

been recently introduced in [16, 6]. They are based on finite interactive systems, a 2-dimensional version of finite automata. Using register machines and space-time duality, the formalism responds to the growing need of programming and reasoning about interactive systems. Its semantics is given in terms of scenarios, built up on top of 2-dimensional words.

The set of contours is enriched with a collection of composition operators. The obtained formalism allows defining a new type of regular expressions over 2-dimensional words n2RE [2], similar to 1-dimensional Kleene formalism. Many interesting open problems naturally occurs in this new formalism n2RE. Here, we are dealing with the formal representation of contours and efficient algorithms for contour representation correctness.

The contribution of this paper is twofold. First, we describe a formal representation of two-dimensional patterns based on contours and their compositions. Then, we present efficient algorithms to verify correctness of the contour-representation.

## 2. ARBITRARY SHAPES IN THE 2-DIMENSIONAL PLANE

2.1. **Contours.** A *(pointed) contour* is a closed, non-overlapping line on a rectangular grid, $\mathbb{Z} \times \mathbb{Z}$, with a chosen start point, surrounding a finite internal area. Each of its segments will be represented using a letter from the set $\{u, d, r, l\}$ ($u$ stands for "up", $d$ for "down", $r$ for "right", and $l$ for "left"), followed by a number denoting its length. A few examples of contours are shown in Fig. 1.

A contour encloses disjoint interior *components*, linked via empty shapes as $rrudll$; the (sub)contours surrounding empty shapes and travelling into the internal area are named *tunnels*, while those in external areas are called *bridges*. A clockwise traversal determines the 2-dimensional area associated to a contour. The area on the east side of a $u$ move is internal, while the one on the west is external. Similar conventions hold for $r, d$ and $l$. Multiple surrounding of the same zone as well as infinite internal areas are forbidden.

Two contours are equivalent iff they enclose the same internal area, modulo translations. For instance, a different placement of the start point determines an equivalent circularly shifted representation. Two equivalent contours are $rrdddllurulldrdlluuurr$ (shortly written as $r^2d^3l^2urul^2drdl^2u^3r^2$) and $d^2l^2urul^2drdl^2u^3r^4d$ - they are presented in Fig. 1(a),(b).

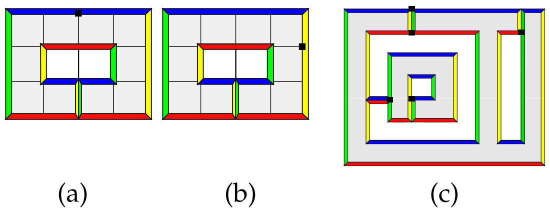

(a) (b) (c)

FIGURE 1. Contours

By filling the interior area of a contour with letters from a given alphabet one gets a *general 2-dimensional word*.

In preparation for the forthcoming formal definition of a contour, some more notations are needed.

*Line segments:* For a vertical line segment $l = ((x, y), (x, y + 1))$ (simply denoted as $l = (x, y + 0.5))$, we denote by $l^k_C$ a predicate which is true if and only if the difference between the "up" and "down" arrows of $C$ passing over $l$ is $k$; notice that $k \in \mathbb{Z}$. A similar notation is used for horizontal line segments $l = ((x, y), (x + 1, y))$.

*Cells:* For a cell $\{(x,y),(x+1,y),(x+1,y+1),(x,y+1)\}$, represented by its center point $c=(x+0.5,y+0.5)$, we denote by $c^k_{C,w}$ a predicate which is true if and only if $l^k_C$ is true, where $l$ is the first line covered by $C$ and situated on the west side of $c$; in this counting bridges and tunnels (lines having equal up/down passings) are skipped. Formally, let

$z=max\{w \in \mathbb{Z} : w \leq x$ and $(l=(w,y+0.5)$ is such that $l^k_C$ is true for a $k\neq 0)\}$;

then $c^k_{C,w}=lz^k_C$, where $lz=(z,y+0.5)$.

*Internal points:* A cell $c$ is *internal from a west perspective* if $c^k_{C,w}$ is true for a $k>0$, meaning there are more $u$ than $d$ passings of the first line segment found towards the west, ignoring bridges and tunnels. The notations $c^k_{C,e}$, $c^k_{C,n}$, and $c^k_{C,s}$ are similarly used to define internal cells from the other perspectives, i.e., examining respectively the est, the north, and the south neighbourhood. The set of cells which are internal to a contour $C$ from all directions is denoted by $Int(C)$.

*External points:* A cell is external from a west perspective if, going towards west, there is no segment with unbalanced $u/d$ moves travelled by $C$ or there is a first line segment with unbalanced $u/d$ passings via $C$ and having more $d$ than $u$ moves. Formally, for all $k$, either $c^k_{C,w}$ is false or $c^k_{C,w}$ is true for a $k<0$. $Out(C)$ denotes the set of cells which are external from all directions.

A contour is well-defined if the set of interior cells $Int(C)$ is finite and the intersection $Int(C)\cap Out(C)$ is empty. The precise definition is described below.

**Definition 2.1.** (a) A string over $\{u,d,l,r\}$ represents a *valid contour* if it describes a closed line and any cell is either internal from all directions or external from all directions; moreover, for all internal cells $C$, all $c^k_{C,w}, c^k_{C,n}$ are satisfied with $k=1$ and all $c^k_{C,e}, c^k_{C,s}$ are satisfied with $k=-1$.

(b) Two contours $C1$ and $C2$ are considered *equivalent* if and only if $Int(C1)=Int(C2)$.

In order to avoid overlapping, each internal cell is surrounded only once. For instance $rdlurdlu$ in not a valid contour, while $rdlu$ is. This shows why in the above definition $|k|$ is restricted to be $1$.

When deciding if a string represents a valid contour, it is useful to have a set of criteria dealing with contour segments, not with the cells. Indeed, inspecting all cells in the grid can be algorithmically inefficient. The equivalent definition in Prop. 2.1 below will be used by the next section algorithms to check if a contour is well defined. Finiteness of $Int(C)$ is equivalent to conditions 2.1 and 2.2. Conditions 2.3 and 2.4 ensure that $Int(C)\cap Out(C)$ is empty.

**Proposition 2.1.** *Let $lv$ (resp. $lh$) denote vertical (resp. horizontal) line segments of a string $C\in\{u,d,r,l\}^*$ enriched with a start point. Then, $C$ represents a valid contour if and only if the following conditions are satisfied:*

**(closed line):**

$$\sum_{lh^k_C} k = 0 \quad and \quad \sum_{lv^k_C} k = 0 \tag{2.1}$$

**(closed shape):**

$$\begin{aligned} &\forall x:\ let\ y_x = min\{y\in\mathbb{Z} : \exists l=(x+0.5,y)\ such\ as\ l^k_C\ is\ satisfied\ for\ a\ k\neq 0\};\\ &if\ y_x\neq nill\ then\ the\ corresponding\ l^k_C\ is\ true\ for\ a\ k<0.\end{aligned} \tag{2.2}$$

**(no repetitions):**

$$\forall l:\ if\,(l=(x,y+0.5)\ or\ l=(x+0.5,y))\ and\ l^k_C\ is\ true\ for\ a\ k\neq 0,\ then\ k\in\{1,-1\}. \tag{2.3}$$

**(alternation in-out):**

*for any pair* $l_1 = (x+0.5, y_1), l_2 = (x+0.5, y_2)$ *of consecutive horizontal borders*

(2.4) $\quad$ (*that is,* $\forall l = (x+0.5, y)$ : *if* $y_1 < y < y_2$, *then* $(l_C^k$ *true* $\Rightarrow (k = 0)))$

*we have:* $if (l_1)_C^{k_1}$ *and* $(l_2)_C^{k_2}$ *are true, then* $k_1 + k_2 = 0$.

*Comments:* Condition 2.1 says the number of left moves equals the number of right moves; and similarly for the vertical direction.

By 2.2, the horizontal line segment with the lowest $y$ coordinate (a line segment situated to the extreme south border) must be oriented left to right. This condition ensures the internal area be finite. For instance $drul$ is not a valid representation, violating this condition. Equivalent presentations of this condition may be introduced using the other directions.

Condition 2.3 has easy intuitive meaning: a contour has no repeated parsing on the borders of a non-empty internal area.

Finally, 2.4 says a contour has no self-intersection, except for tangential contact of disjoint areas, namely tunnels. Horizontal segments with unequal $r/l$ passing, situated at the same $x$ coordinate, must alternate $r/l$ directions (i.e., the difference $r - l$ is a sequence $-1, 1, -1, \ldots$ or $1, -1, 1, \ldots$). This ensures each cell belongs either to $Int(C)$ or to $Out(C)$.

In Fig. 2 some examples of invalid contour representations are illustrated: (a) $ldru$, (b) $rdlururd^3lulu$, (c) $r^4d^3l^2ulur^2dldl^2u^3$, and (d) $r^2d^3l^3u^2l^2dlu^2$. The contour in (a) is not representing a finite shape; the contour in (b) is passing the cell $(1.5, 1.5)$ twice; the cells $(1.5, 1.5)$ and $(2.5, 1.5)$ belong either to the interior or to the exterior area of the contour in (c); finally, the contour in (d) is self-intersecting.

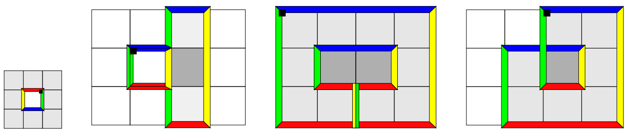


FIGURE 2. Examples of invalid contours

2.2. **A normal form representation of contours.** A string $C$ is called a *simple contour* if it represents a contour and there are no proper substrings of $C$ with this property. It represents a finite area with no holes. The representation of a simple contour is unique modulo the position of the start point.

Two *edge-neighbouring cells* are two cells which share a horizontal or a vertical edge. An *edge-connected component* is a maximal set of cells such that any two cells are connected with a path of edge-neighbouring cells, all from that component.

A (general) contour may be decomposed such that each of its *edge-connected* components is represented by a simple contour (used for its external border) and zero, one or more "inverses" of simple contours for its possible internal holes. This decomposition, called the *normal form representation* of a contour, is unique up to connecting identities. (Recall that identities are contours with empty interior area, e.g., tunnels or bridges.)

For instance the contour: $r^5dld^5ru^6rd^7l^9u^7r^3dl^2d^3ru^2r^3d^3l^2uruld^2luld^2r^5u^5l^3u$, depicted in Fig. 1(c), may be decomposed into two connected components and has the following normal form representation:

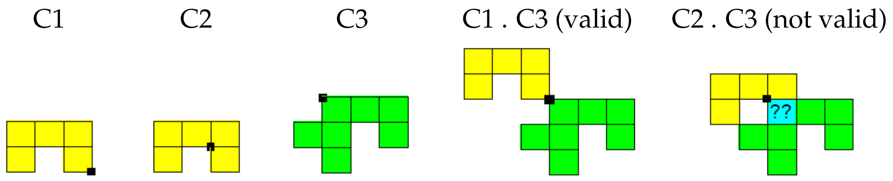
The *inverse* of a string C over the alphabet $\{u, d, l, r\}$ is obtained by replacing $u/d/l/r$ with $d/u/r/l$, respectively.

- the components: (a) {contour $\{r^9d^7l^9u^7\}$ and holes $\{l^5d^5r^5u^5\}, \{ld^5ru^5\}\}$; and (b) {contour $\{r^3d^3l^3u^3\}$ and holes $\{ldru\}\}$;
- the information on the relative position of the internal holes in the surrounding contours given by connecting identities (not shown here).

As illustrated by the example above, the normal form of a contour alternates the traversals of exterior and interior shapes. The only variation are the bridges/tunnels connecting simple contours.

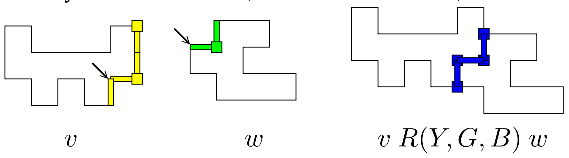
2.3. **2-dimensional regular expressions.** The set of well-defined contours is enriched with a binary composition operator "$.$": the result of composing $C1$ and $C2$ is the string $C1\,C2$, provided this is a valid contour. This means, the contours are glueing together via the starting points used in their representations.

For a graphical example, notice that C1 . C2 below shows a valid composition, while C2 . C3 shows an example of composition leading to an invalid result (the result has overlapping areas).



*Generic definition for restricted composition operators:* Restricted composition operators are obtained from the following generic format. Suppose we are given:

(1) 2 words $v, w$; a subset $Y$ of elements of the contour of $v$ (the yellow elements in the figure below); a subset $G$ of elements of the contour of $w$ (the green ones); the subset $B$ of actual contact elements after composing, as above, $v$ with $w$ via the points indicated by a little arrow (the blue elements).



(2) a relation $R(Y, G, B)$ between the above 3 subsets.

The resulted restricted composition is denotes by $v\ R(Y, G, B)\ w$.

In the given example, a relation $R$ making the restricted composition valid may be: $G \subseteq Y \wedge G \subseteq B$ (after composition, all the elements in the green set are on the common border and included in the yellow set).

*A set of particular restricted composition operators n2RE:* A line $l = (x, y + 0.5)$ is on the *east border* of a contour $C$ if $l_C^k$ holds for $k = -1$; equivalently, the cell $(x - 0.5, y + 0.5)$ is internal, while $(x + 0.5, y + 0.5)$ is in the exterior area. Similarly, a point $p = (x, y)$ is on the south-east border of $C$ if the cell $c = (x - 0.5, y + 0.5)$ is in the internal area of $C$, while the other 3 cells around are in the external area of $C$. Bridges or tunnels are not be counted as borders.

Let us use the following notation: $w$ for "west border", $e$ for "east border", $n$ for "north border", $s$ for "south border", $nw$ for "north-west point", $ne$ for "north-east point", $sw$ for "south-west point", and $se$ for "south-east point". We denote by $Connect$ their set $\{w, e, n, s, nw, ne, sw, se\}$.

On each of the above eligible glueing combination $(x, y) \in Connect$ we put a constrain consisting of a propositional logic formula $F \in PL(\phi_1, \phi_2, \phi_3, \phi_4)$ , i.e., a boolean formula built up starting with the following atomic formulas:

$\phi_1(x, y) =$ "$x < y$", $\phi_2(x, y) =$ "$x = y$", $\phi_3(x, y) =$ "$x > y$", and $\phi_4(x, y) =$ "$x$ # $y$".

The meaning of the connectors is the following: "<" - left is included into the right; "=" - left is equal to the right; ">" - left includes the right; "$x$ # $y$" - left and right overlaps, but none is included in the other.

For instance: $f(e = w)g$ means "restrict the general composition of $f$ and $g$ such that the east border of $f$ is identified to the west border of $g$"; $f(e > w)g$ - the east border of $f$ includes all the west border of $g$, but some east borders of $f$ may still be not covered by west borders of $g$; etc.

We also use the notation

$\phi_0(x, y) =$ "$x$ $O$ $y$", where "$O$" means empty intersection.

Actually, this is a derived formula $\neg(\phi_1(x, y) \vee \phi_2(x, y) \vee \phi_3(x, y) \vee \phi_4(x, y))$.

**Definition 2.2.** (restricted compositions) A *restriction formula* $\phi$ is a boolean combination in $PL(F_1, \ldots, F_n)$, where $F_i$ are constricting formulas involving certain eligible glueing combinations $(x_i, y_i) \in Connect$. A *restricted composition operation* $\_(F)\_$ is the restriction of the general composition to composite words satisfying $F$. A word $h \in f\ .\ g$ belongs to $f\ (F)\ g$ if for all glueing combinations $(x_i, y_i)$ occurring in $F$ the contact of the $x_i$ border of $f$ and $y_i$ border of $g$ satisfies $F_i$. *Iterated composition operators* are denoted by $*(F)$, for a restriction formula $F$. □

The class of *new regular expressions* for 2-dimensional in [2], denoted n2RE, consists of all expressions obtained using the operators introduced in this paragraph.

*Example:* An example of n2RE expression (for spiral words

```
x     2aa     2aaaa
      2x1     22aa1
      bb1     22x11
              bbbb1
```

...) is:

```
x(e<w & w<e & n<s & s<n)
{[R(se>ne)D](nw<ne & sw>se)[L(nw>sw)U]}*_(e<w & w<e & n<s & s<n)
```

where `R = a*_(e.w),  D = 1*_(s.n),  L = b*_(e.w),  U = 2*_(s.n).`

## 3. Algorithms for testing correct representations of shapes

In this section we presents two procedures for verifying the correctness of contour representations.

Dealing with cell criteria stated in Def. 2.1 may lead to inefficient algorithms. Basically, in order to decide if a given sting $C \in \{u, d, r, l\}^*$ is a valid contour, one has to determine the membership of each cell either to $Int(C)$ or $Out(C)$ by calculating the predicates $c^k_{C,w}, c^k_{C,n}, c^k_{C,e}, c^k_{C,s}$. Thus, the advantage of having a 1-dimensional string representation is not exploit, as the analysis deals with the full 2-dimensional plane.

The following algorithms are based on the equivalent conditions in Prop. 2.1, dealing with contour segments. Testing if a contour is a closed line can be easily done in linear time; the main difficulty remains to verify conditions 2.2, 2.3, and 2.4.

### 3.1. The 1st algorithm for valid contours.

The first algorithm is based on sorting, achieving $O(n \log n)$ complexity, where $n$ is the contour length. The list $l[1 \ldots n]$ of lines associated with the letters of the contour is sorted according to $(x, y)$ coordinates. The values $k$ that satisfy the predicates $l[i]^k_C$ are calculated in one traversal of the sorted list, as the informations to be added for each segment are situated on consecutive positions. Further

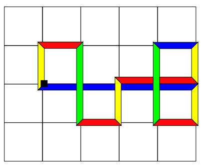
*PL*(*Atom*) denotes the set of propositional logic formulas built up with atomic formulas in *Atom*. For typing reasons, the boolean operations "not", "and", and "or" are denoted by "!", "&", and "V", respectively.

checking of requirements 2.2, 2.3, and 2.4 is immediate. The full description of this 1st algorithm is shown in Fig. 3 and an example in Example 3.1. It takes as input the string denoting the moves along the contour to be verified; the stating point is set to $(0, 0)$ (the contours are invariant to translations, so the starting point position does not matter).

```
1: function TESTCONTOUR(char C[1 ...n])
2:    int x, y, m; float aux[1 ...n][1 ...3], int k, prec; boolean valid;
3:    x := 0, y := 0;
4:    for k:=1 to n
5:      switch C[k]
6:       case 'r':  aux[k][1] := x + 0.5; aux[k][2] := y; x := x+1;
7:       case 'd':  aux[k][1] := x; aux[k][2] := y - 0.5; y := y-1;
8:       case 'l':  aux[k][1] := x - 0.5; aux[k][2] := y; x := x-1;
9:       case 'u':  aux[k][1] := x; aux[k][2] := y + 0.5; y := y+1;
10:    Sort(aux);
11:    k:= 1; m:=0; valid:= true; prec := 0;
12:    while k ≤ n
13:     x := aux[k][1]; y := aux[k][2]; m:= m+1;
14:     l[m][1]:=x; l[m][2]:=y; l[m][3]:=0;
15:     while k ≤ n ∧ x = aux[k][1] ∧ y = aux[k][2]
16:      switch C[aux[k][3]]
17:       case 'r':  l[m][3] := l[m][3] + 1;
18:       case 'd':  l[m][3] := l[m][3] - 1;
19:       case 'l':  l[m][3] := l[m][3] - 1;
20:       case 'u':  l[m][3] := l[m][3] + 1;
21:      k := k+1 ;
22:     If l[m][3] < -1 ∨ l[m][3] > 1
23:      valid := false;
24:     If l[m][3] <> 0 ∧ prec <> 0 ∧ x = l[prec][1]
25:      If l[prec][3] + l[m][3] <> 0
26:       valid := false;
27:      prec:=m;
28:     If (prec = 0 ∧ round(x) <> x) ∨
29:            (prec <> 0 ∧ x <> l[prec][1] ∧ round(x) <> x)
30:      If l[m][3] > 0
31:       valid := false;
32:      If l[m][3] <> 0
33:       prec := m;
34:    return valid;
35: end function
```

FIGURE 3. Algorithm for checking the correctness of contour representations

**Example 3.1.** Taking as input the contour $C = rrrrdluurdlldluuld$



the algorithm described in Fig. 3 runs as follows.

1. First it calculates the segments reached by $C$ and stores the result in `aux` (lines 4-9 in Algoritm 3).

$$(0,0) \xrightarrow[(0.5,0)]{1:r} (1,0) \xrightarrow[(1.5,0)]{2:r} (2,0) \xrightarrow[(2.5,0)]{3:r} (3,0) \xrightarrow[(3.5,0)]{4:r} (4,0) \xrightarrow[(4,-0.5)]{5:d} (4,-1) \xrightarrow[(3.5,-1)]{6:l}$$
$$(3,-1) \xrightarrow[(3,-0.5)]{7:u} (3,0) \xrightarrow[(3,0.5)]{8:u} (3,1) \xrightarrow[(3.5,1)]{9:r} (4,1) \xrightarrow[(4,0.5)]{10:d} (4,0) \xrightarrow[(3.5,0)]{11:l} (3,0) \xrightarrow[(2.5,0)]{12:l}$$
$$(2,0) \xrightarrow[(2,-0.5)]{13:d} (2,-1) \xrightarrow[(1.5,-1)]{14:l} (1,-1) \xrightarrow[(1,-0.5)]{15:u} (1,0) \xrightarrow[(1,0.5)]{16:u} (1,1) \xrightarrow[(0.5,1)]{17:l} (0,1) \xrightarrow[(0,0.5)]{18:d}$$
$$(0,0)$$

2. The second step is to sort the list above, according to $(x,y)$ coordinates:

$(0,0.5,\overset{18}{d})$ $(0.5,0,\overset{1}{r})$ $(0.5,1,\overset{17}{l})$ $(1,-0.5,\overset{15}{u})$ $(1,0.5,\overset{16}{u})$ $(1.5,-1,\overset{14}{l})$ $(1.5,0,\overset{2}{r})$ $(2,0.5,\overset{13}{d})$ $(2.5,0,\overset{3}{r})$ $(2.5,0,\overset{12}{l})$ $(3,-0.5,\overset{7}{u})$ $(3,0.5,\overset{8}{u})$ $(3.5,-1,\overset{6}{l})$ $(3.5,0,\overset{4}{r})$ $(3.5,0,\overset{11}{l})$ $(3.5,1,\overset{9}{r})$ $(4,-0.5,\overset{5}{d})$ $(4,0.5,\overset{10}{d})$

3. The final loop (lines 17-20) calculates how many times each segment is passed by the contour (i.e., the value $k$ for which the predicate $l^k_C$ is true is calculate for each line `l[m]`). For example, the line $l = (2.5,0) = ((2,0),(3,0))$ is a bridge, being crossed by an equal number of left and right moves; this information is calculated adding positions 9 and 10 in the sorted list.

4. Conditions 2.2, 2.3 and 2.4 are checked at lines 28-31, 22-23 and 24-27 respectively. □

With minor adjustments, the procedure above may generate a normal form of the input contour.

3.2. **The 2nd algorithm - an optimized version.** This version is an optimized version of the 1st algorithm. It applies a computational geometry technique (line sweeping) adapted to the set of segments composing a contour. Each letter repetition denotes a longer vertical or horizontal segment with a given orientation and length. The time complexity of the algorithm reduces to $O(nr \log(\max y))$, where $nr$ is the number of segments and $\max(y)$ is the difference between the greatest and the lowest $y$ coordinate reached.

*Sweep line or sweep surface* is a common concept in geometric algorithms. Usually, it consists in a vertical imaginary line that moves across the plane and stops in certain points, where its state is changed. The solution is found after all the stop points (events) are processed, gathering informations from all the neighboring objects.

In the case of a contour, *stop points*, sorted in increasing order, are all $x$-coordinates of the composing segments. Hence there are three possible events:

- left margin $(x1,y)$ of a horizontal segment ($l$ or $r$) $\Rightarrow$ $Update(y,+1)$;
- right margin $(x2,y)$ of a horizontal segment $\Rightarrow$ $Update(y,-1)$;
- vertical segment $(x,y1,y2)$ ($u$ or $d$) $\Rightarrow$ $Query(y1,y2)$.

A balanced binary tree may store the *line state*. Each node corresponds to an interval $[y1,y2]$ with offspring $[y1,(y1+y2)/2]$ and $[(y1+y2)/2+1,y2]$. The information $AUX[y1,y2]$, memorized as a heap, indicates the number of horizontal arrows intersected by the sweep line between $y1$ and $y2$, meaning how many $(x1,y), y \in [y1,y2]$ were swept without the corresponding $(x2,y)$ to by reached. Events of first and second type update the interval tree, adding or subtracting 1 to a certain leaf $AUX[k]$, where $k$ is the heap index corresponding to an interval of size 0, $[y,y]$. $AUX[k][0], AUX[k][1]$ counts the number of segments oriented left and right, respectively, reaching the coordinate $y$. All nodes on the path from root (corresponding to the interval $[0,max(y)]$), to the leaf $k$ are updated.

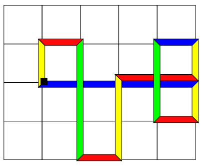
The algorithm detects possible self-intersections when reaching a vertical segment $[y1,y2]$. By questioning the line state it verifies that no horizontal segments lies between $[y1+1,y2-1]$. $Query(y1,y2)$ is a divide and conquer procedure that sums informations found in the set of vertices composing a minimal partition of the segment $[y1, y2]$. If a query returns at least 1 then the contour is self intersecting.

The length of the root interval of the balanced tree, determines the complexity of each update and query operation: $O(\log(\max(y))$. As the number of events can't exceed twice the number of segments, the overall complexity is $O(nr\log(max(y))$.

```
function QUERY(int index, int rootLeft, int rootRight, int qLeft, int qRight)
  If rootLeft ≥ qLeft and rootRight ≤ qRight
    return abs(AUX[index][0] - AUX[index][1]);
  int resultLeft = 0, resultRight = 0, m = (rootLeft+rootRight)/2
  If leftQ ≤ m
    resultLeft = Query(index*2,rootLeft,m,qLeft,qRight);
  If rightQ > m
    resultRight = Query(index*2+1,m+1,rootRight,qLeft,qRight) ;
  return (resultLeft + resultRight) ;
end function
```

FIGURE 4. An optimized version of the Algorithm in Fig. 3 - the key function

**Example 3.2.** The contour $C = r^4dlu^2rdl^2d^2lu^3ld$



generates the list of events

$U(2, +l)$, $U(3, +r)$, $U(0, +l)$, $Q(1, 2) = 1$, $U(3, -l)$ , $U(2, +l)$, $Q(1, 1) = 0$, $U(0, -l)$, $U(1, +l)$, $U(3, +r)$, $Q(2, 2) = 0$, $U(1, -l)$, $U(3, -l)$.

The first query event corresponds to the first vertical segment, considering that segments are ordered according to $x$ coordinate. At this point, the state of the sweep-line memorize one horizontal segment oriented to right, with $y$ in the query interval $[1, 2]$. Hence $Q(1, 2) = 1$. □

3.3. **Comparison of the algorithms.** For the comparison of the the two algorithms presented above, we have performed tests on nine sets of randomly generated contours, of various length and shapes.

We first analyse the performance on dense contours, consisting in one edge-connected component. The average execution time, in seconds, for each set of valid dense contours is presented in Table 1. The last column shows, in percent, the time of the 2nd algorithm compared to the 1st (the smallest the numbers, the better the 2nd algorithm performance).

The results obtained for sparse contours is summarized in Table 2. Sparse contours are valid composites of small rectangular contours with identities of various length. It may be seen that better execution times are obtained with the 2nd algorithm in case of contours with large distances between components or with components represented with large segments.

Finally, results for composed contours mixing the above two sets are shown in Table 3. The notation DENSE$i$/SPARSE$i$ refers to the $i$-th line in the DENSE/SPARSE table above.

| *DENSE* | Average execution time | | time T2 / time T1 (%) |
|---|---|---|---|
| contour length | first algorithm T1 | second algorithm T2 | T2*100/T1 |
| $< 10000$ | $0,3572$ | $0,1049$ | $29,8947$ |
| $> 10000$ | $2,0573$ | $0,4015$ | $20,1360$ |
| $< 500$ | $0,0688$ | $0,0667$ | $94,3219$ |

TABLE 1. Comparison of the algorithms on dense contours

| *SPARSE* | | Average execution time | | time T2 / time T1 (%) |
|---|---|---|---|---|
| cells | distance | first algorithm T1 | second algorithm T2 | T2*100/T1 |
| $[100,200]$ | $[25,50]$ | $2,6484$ | $1,2521$ | $47,3936$ |
| $[200,300]$ | $[25,50]$ | $3,7114$ | $1,7215$ | $46,4073$ |
| $[100,200]$ | $[50,100]$ | $7,1069$ | $2,5222$ | $35,5811$ |

TABLE 2. Comparison of the algorithms on sparse contours

| *COMPOSED* | | Average execution time on c1.c2 | | time T2 / time T1 (%) |
|---|---|---|---|---|
| c1 | c2 | first algorithm T1 | second algorithm T2 | T2*100/T1 |
| DENSE1 | SPARSE1 | $2,9360$ | $0,9170$ | $31,3385$ |
| DENSE2 | SPARSE2 | $6,9120$ | $1,7882$ | $26,2817$ |
| DENSE3 | SPARSE3 | $5,8749$ | $1,6525$ | $28,2066$ |

TABLE 3. Comparison of the algorithms on composed contours

We mention that in both implementations, we stop when one of the conditions 2.2, 2.3 or 2.4 is not checked. Hence, for invalid contours, the performance my also depend on the start point, the place where the first self-intersection is placed, etc.

## 4. CONCLUSIONS AND FUTURE WORKS

The known approach [8] to get regular expressions for 2-dimensional patterns uses intersection and renaming - see [12, 2] for some critics on using these operators. One of the benefits of our approach here and of the new type of regular expressions n2RE introduced in [2] is that renaming and intersection are avoided, the setting being closer in spirit with classical 1-dimensional regular expressions.

Current hardware and software development, mainly driven by multi-core architectures and distributed computing technologies, bring forward the necessity to adapt sequential machine models to interactive computation. The results in this paper are steps of a program extending sequential computation models in this direction.

As future work we intend to prove a Kleene theorem for finite interactive systems and also to develop an associated algebraic theory, similar to automata theory. Possible applications of the model are: image processing and image recognition procedures, study of parallel, interactive OO-programs, modelling discrete physical or biological systems etc.

## REFERENCES

[1] Asarin, E., Caspi, P., and Maler, O., *Timed regular expressions*, Journal of the ACM, **49** (2002), 172–206

[2] Banu-Demergian, I. T., Paduraru, C. I., and Stefanescu, G., A new representation of two-dimensional patterns and applications to interactive programming, in *Proceedings FSEN 2013*, LNCS 8161, Springer, 2013, 183–198

[3] Bentley, J. and Ottmann, T. A., *Algorithms for reporting and counting geometric intersections*, IEEE Transactions on Computers, **100** (1979), 643–647

[4] Bribiesca, E. and Verlade, C., *A formal language approach for a 3D curve representation*, Computers & Mathematics with Applications, **42** (2001), 1571–1584

[5] Conway, J. H., *Regular Algebra and Finite Machines*, Chapman and Hall, 1971

[6] Dragoi, C. and Stefanescu, G., *AGAPIA v0.1: A programming language for interactive systems and its typing systems*, ENTCS, **203** (2008), 69–94

[7] Freeman, H., *On the encoding of arbitrary geometric configurations*, IRE Transactions on Electronic Computers, **2** (1961), 260–268

[8] Garg, V. and Ragunath, M. T., *Concurrent regular expressions and their relationship to Petri nets*, Theoretical Computer Science, **96**(1992), 285–304

[9] Giammarresi, D. and Restivo, A., Two-dimensional languages, in *Handbook of Formal Languages. Vol.* **3***: Beyond Words*, Springer-Verlag, 1997, 215–265

[10] Goldin, D., Smolka, S., and Wegner, P., Eds., *Interactive Computation: The New Paradigm*, Springer, 2006

[11] Kleene, S. C., Representation of events in nerve nets and finite automata, in *Automata Studies*, Princeton University Press, 1956, 3–41

[12] Kozen, D., A completeness theorem for Kleene algebras and the algebra of regular events, in *Proceedings LICS 1991*, 214-225

[13] Kaneko, T. and Okudaira M., *Encoding of arbitrary curves based on the chain code representation*, IEEE Transactions on Communications, **33** (1985), 697–707

[14] Liu, Y. K. and Zalik, B., *An efficient chain code with Huffman coding*, Pattern Recognition, **38** (2005), 553–557

[15] Stefanescu, G., Algebra of networks: Modeling simple networks as well as complex interactive systems, in *Proof and System Reliability*, Kluwer, 2002, 49–78

[16] Stefanescu, G., *Interactive systems with registers and voices*, Fundamenta Informaticae, **73** (2006), 285–306

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF BUCHAREST, ROMANIA
*E-mail address*: `iulia.banu@fmi.unibuc.ro`
*E-mail address*: `gheorghe.stefanescu@fmi.unibuc.ro`