A linear time and space algorithm for detecting path intersection ☆

Srečko Brlek[a], Michel Koskas[b], Xavier Provençal[c]

[a] *Laboratoire de Combinatoire et d'Informatique Mathématique, Université du Québec à Montréal, C. P. 8888 Succursale "Centre-Ville", Montréal (QC), CANADA H3C 3P8*
[b] *UMR AgroParisTech/INRA 518 16 rue Claude Bernard 75 231 Paris Cedex 05*
[c] *LAMA, CNRS UMR 5127, Université de Savoie, 73376 Le Bourget-du-lac cedex.*

Abstract

The Freeman chain is a common encoding of discrete paths by means of words such that each letter encodes a step in a given direction. In the discrete plane $\mathbb{Z}^2$ such a coding is widely used for representing connected discrete sets by their contour which forms a closed and intersection free path. In this paper, we use of a multidimensional radix tree like data structure for storing paths in the discrete d-dimensional space $\mathbb{Z}^d$. It allows to design a simple and efficient algorithm for detecting path intersection. Even though an extra initialization is required, the time and space complexity remain linear for any fixed dimension d. Several problems that are solved by adapting our algorithm are also discussed.

Keywords: Freeman code, lattice paths, self-intersection, radix tree

1. Introduction

In dimension two, a finite and connected discrete set $S \subset \mathbb{Z}^2$ is represented by a discrete contour path L. In such case, many geometrical properties of S can be computed from its contour word w in time linear in its length $|w| = n$. Among the many problems that have been considered in the literature, we mention: computations of statistics such as area, moment of inertia [4, 5], digital convexity [6, 7, 8], and tiling of the plane by translation [9, 10]. In all of the above mentioned problems, the fact that such a word is a contour word is usually assumed a priori. In $\mathbb{Z}^d$, paths are conveniently encoded by words on the alphabet $\Sigma_d = \{a_1, \ldots, a_d, \overline{a}_1, \ldots, \overline{a}_d\}$. The problem of determining whether a given word w is a contour word amounts to check first that w encodes a closed path (*i.e.* starting and ending at the same point), and secondly that it is not self-intersecting (*i.e.* never visiting some point twice). Testing that w encodes a closed path is easily obtained by checking that the number of occurrences in w of each elementary step a_k is equal to its opposite one $\overline{a}_k$, for all $k = 1, \ldots, d$.

☆with the support of NSERC (Canada)

Email addresses: Brlek.Srecko@uqam.ca (Srečko Brlek), michel.koskas@agroparistech.fr (Michel Koskas), xavier.provencal@univ-savoie.fr (Xavier Provençal)

On the other hand, testing that it does not intersect itself requires more work. Indeed, it requires to check that no grid point is visited twice, for which one might easily design an $\mathcal{O}(n \log n)$ algorithm where sorting is involved, or use hash tables that provide a linear time algorithm on the average but not in the worst case.

This major drawback was removed in [1] by providing a linear time and space algorithm checking if a path in $\mathbb{Z}^2$ encoded by a word on the alphabet $\{a_1, a_2, \overline{a}_1, \overline{a}_2\}$ visits any of the grid points twice. In this extended version, we present a generalization of this algorithm to paths in any arbitrary dimension $d \geq 2$. Although most of the illustrations provided are in dimension two for practical reasons, the data structure is completely generalized to higher dimensional spaces. Following [1], we first describe the data structure in the simple case where points have only positive coordinates, and then, we adapt it in order to deal with negative coordinates, in a new and more efficient way.

The paper is structured as follows. We begin by presenting in Section 2 a data structure based on a multidimensional quadtree [11] used to represent points of the space $\mathbb{N}^d$. This tree is labeled in order to provide a radix-tree (see for instance [12]) structure that allows to represent any finite subset of $\mathbb{N}^d$. Then, in Section 3 this structure is enriched with *neighbor links* in order to provide fast access to nodes representing points that are close to each other. Section 4 decribes the algorithms in detail, and its complexity analysis is carried out in Section 5. Since the data structure presented represents points of $\mathbb{N}^d$ only, its extension to $\mathbb{Z}^d$ is more involved and detailed in Section 6. In Section 7 performance and comparison with other classical algorithms are discussed, along with experimental results. Due to its universality, this algorithm has many applications and we show in Section 8 how it can be used/modified for efficiently (e.g. in linear time) solving problems like determining if a word is the contour of a discrete figure, the nature of intersections, the multiplicity of each visited grid point and multiple multiple paths intersection.

2. Radix tree structure

In a first stage, we build a multidimensional radix tree for representing subsets of $\mathbb{Z}^d$. This structure enables us to represent any given subset $X \subset \mathbb{Z}^d$ with a worst-case time and space complexity of $O(n \log m)$ where $n = |X|$ and $m = \max\{||x||_\infty | x \in X\}$ where $||x||_\infty$ is the ∞-norm of x, that is, the greatest of its coordinates in absolute value. Later on, this structure is enriched by using the so-called *neighbor links* which are essential for reducing to a linear time and space complexity in worst-case when the subset X is given as a discrete path.

As mentioned previously, for now we focus our attention on points in $\mathbb{N}^d$, that is, having only nonnegative coordinates represented as binary strings of arbitrary length. Let $\mathbb{B} = \{0, 1\}$ be the base for writing integers. Words in $\mathbb{B}^*$ are conveniently represented in the *radix order* by a complete binary tree (see for instance [12, 13]), where the level k contains all the binary words of length k, and the order is given by the breadth-first traversal of the tree. To distinguish a natural number $x \in \mathbb{N}$ from its representation we write $\boldsymbol{x} \in \mathbb{B}^*$. The edges are defined inductively by the rewriting rule $\boldsymbol{x} \longrightarrow \boldsymbol{x} \cdot 0 + \boldsymbol{x} \cdot 1$, with the convention that 0 and 1 are the labels of, respectively, the left and right edges of the node having value $\boldsymbol{x}$. This representation is extended to $\mathbb{B}^* \times \mathbb{B}^* \times \cdots \times \mathbb{B}^*$ as follows.

A radix tree structure for points in the integer space. As usual, the concatenation of words is extended to the Cartesian product of words by setting for $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_d) \in \mathbb{B}^* \times \cdots \times \mathbb{B}^*$, and $(\alpha_1, \alpha_2, \ldots, \alpha_d) \in \mathbb{B} \times \cdots \times \mathbb{B}$,

$$(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_d) \cdot (\alpha_1, \alpha_2, \ldots, \alpha_d) = (\boldsymbol{x}_1 \cdot \alpha_1, \boldsymbol{x}_2 \cdot \alpha_2, \ldots, \boldsymbol{x}_d \cdot \alpha_d).$$

Let $x = (x_1, x_2, \ldots, x_d) \in \mathbb{N}^d$ and assume that all $\boldsymbol{x}_i$ have same length (provided padding with zeros is applied on the left if necessary). Then the rule

$$x \longrightarrow \sum_{\alpha \in \{0,1\}^d} x \cdot \alpha, \tag{1}$$

defines the quadtree $RT = (N, R)$ such that

(i) the root is labeled $(0, 0, \ldots, 0)$;

(ii) each node (except the root) has 2^d sons;

(iii) if a node is labeled $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_d)$ then $|\boldsymbol{x}_i| = |\boldsymbol{x}_j|$ for all i, j;

(iv) edges are undirected (may be followed in both directions).

By convention, edges leading to the sons are labeled by the corresponding d-tuple from the set $\{0, 1\}^d$. These labels equip the quadtree with a *radix tree* structure for Equation (1) implies that x' is a son of x with label α, if and only if for all $i \in \{1, 2, \ldots, d\}$,

$$x'_i = 2x_i + \alpha_i. \tag{2}$$

Observe that any point $x = (x_1, x_2, \ldots, x_d)$ is represented exactly once in this tree. We denote by $\textcircled{x}$ the node representing the point x. The sequence of d-tuple of digits (the d digits in first place, the d digits in second place, and so on) gives the unique path in the tree leading from the root to this node. Of course, unlike the other nodes, the root may only have up to $2^d - 1$ sons since no edge labeled $(0, 0, \ldots, 0)$ starts from the root.

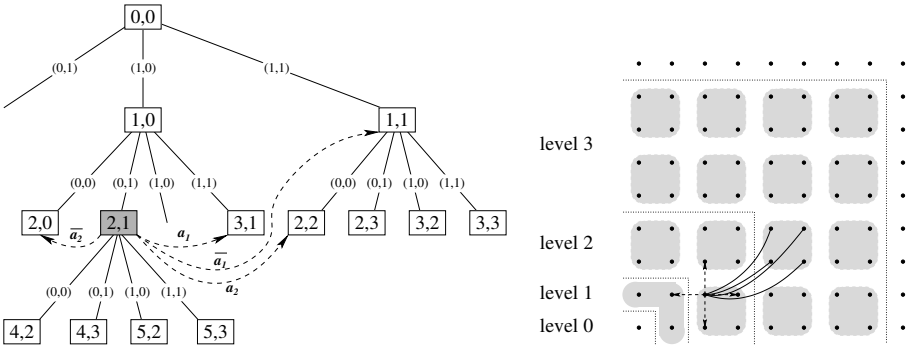

Figure 1: Left: the point $(2, 1)$ with its sons and neighbors in the radix tree. Right: the partition of $\mathbb{N} \times \mathbb{N}$ defined by the radix tree.

Now, in order to store any finite subset $X \in \mathbb{N}^d$ using such a radix tree, it suffices to add a boolean flag on each node indicating that it represents either a point of X or an inner node of the structure. For example, Figure 1 (left) shows

the upper levels of the radix tree used in dimension 2. Note that in order to represent the point $(2,1)$ the tree must contain at least the nodes representing $(0,0)$ and $(1,0)$. On the other hand, no other nodes are required for storing the set $\{(2,1)\}$. As such, the space $\mathbb{N}^d$ is partitioned into layers depending on the length of the binary representation of coordinates $(x_1, x_2, \ldots, x_d)$ of each point. More precisely, for points having nonnegative coordinates, the point at the origin forms the layer 0, while for $n \geq 1$ the n-th layer contains the points such that $n = 1 + \max\{\log_2(|x_i|) \,|\, 1 \leq i \leq d\}$ (See Figure 1, left).

2.1. Dynamic construction of the radix tree

To build the radix tree RT of some subset $X \subset \mathbb{N}^d$, we proceed dynamically by sequential insertion of points starting with the *empty tree*, that is a radix tree consisting of the root only. Of course, in the empty tree, the root's flag is set to `false` so that it represents the empty set.

Algorithm 1: `InsertPoint`

Input: $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_d)$ with each $\boldsymbol{x}_i \in \mathbb{B}^k$ and $RT = (N, R)$;
//Inserts the point x to the radix tree RT;
$\textcircled{n} \leftarrow$ the root of RT;
while $k > 0$ **do**
  $\alpha \leftarrow (x/2^{k-1}) \mod 2$;
  $k \leftarrow k - 1$;
  $\textcircled{n} \leftarrow$ the son of $\textcircled{n}$ labeled by α;
  //If such a node does not exist, it is created
Mark $\textcircled{n}$'s flag as `true`;

In Algorithm 1 we use the convention that all coordinates of x are written as binary words of length k where k is the smallest integer that allows to do so. In this algorithm, the modular operation is meant on d-tuples : "$x \mod 2$" is the d-tuple $(x_1 \mod 2, x_2 \mod 2, \ldots, x_d \mod 2)$.

Clearly, both the time complexity and the space complexity of Algorithm 1 are $O(d \log ||x||_\infty) = O(\log ||x||_\infty)$. In practice, we assume that the dimension d is fixed so that a multiplicative factor in terms of d should be considered as a constant. The time and space complexity of n successive calls to this algorithm is $O(n \log m)$ where m is the average value of the greatest coordinate of each of the n points inserted in RT.

Once a given subset $X \in \mathbb{N}^d$ is stored in such a radix tree, one may easily modify Algorithm 1 in order to test that a given point x is included in X in time $O(\log ||x||_\infty)$.

Finally, remark that one may have to deal with a set of points $X \subset \mathbb{N}^d$ such that all points are relatively close to each other (*i.e.* $\max\{||x - y||_\infty \,|\, x, y \in X\}$ is small) while having large coordinates (*i.e.* $||x||_\infty$ is large for most of the $x \in X$). In this case, a clever use of this data structure would consist in storing the position of these points relatively to some well chosen point $p \in N^d$ such that $||x - p||_\infty$ is small for most of the $x \in X$. The time complexity of inserting n points to RT becomes $O(n \log m')$ where m' is the average value of $||x - p||_\infty$ for all $x \in X$.

3. Paths, words and neighbors

A word w is a finite sequence of letters $w_1 w_2 \cdots w_n$ on a finite alphabet Σ, that is a function $w : [1..n] \longrightarrow \Sigma$, and $|w| = n$ is its *length*. Therefore the ith letter of a word w is denoted w_i, and sometimes $w[i]$ when we emphasize the algorithmic point of view. The empty word is denoted ε. The set of words of length n is denoted Σ^n, that of length at most n is $\Sigma^{\leq n}$, and the set of all finite words is Σ^*, the free monoid on Σ. From now on, the alphabet is fixed to $\Sigma = \Sigma_d\{a_1, a_2, \ldots, a_d, \overline{a}_1, \overline{a}_2, \ldots, \overline{a}_d\}$. To any word $w \in \Sigma^*$ is associated a vector $\overrightarrow{w}$ by the morphism $\overrightarrow{\ } : \Sigma^* \longrightarrow \mathbb{Z}^d$ defined on each letter of Σ by :

$$\overrightarrow{a_i} = e_i, \quad \overrightarrow{\overline{a}_i} = -e_i.$$

where e_i is the i-th vector of the canonical basis of $\mathbb{R}^d$. Then, for all $u, v \in \Sigma^*$, we have $\overrightarrow{u \cdot v} = \overrightarrow{u} + \overrightarrow{v}$.

The set of elementary translations allows to draw each word as a $(2d)$-connected path in $\mathbb{Z}^d$ starting from the origin. In particular, as shown on the

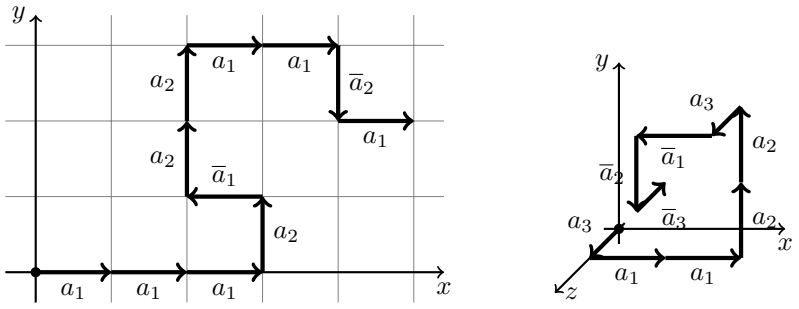

Figure 2: Paths $w_1 = a_1a_1a_1a_2\overline{a}_1\overline{a}_2\overline{a}_2a_1a_1\overline{a}_2a_1$ (left) and $w_2 = a_3a_1a_1a_2a_2a_3\overline{a}_1\overline{a}_2\overline{a}_3$ (right).

left part in Figure 2, a path in $\mathbb{Z}^2$ is coded on the alphabet $\{a_1, a_2, \overline{a}_1, \overline{a}_2\}$ going *right* for a letter a_1, *left* for a letter $\overline{a}_1$, *up* for a letter a_2 and *down* for a letter $\overline{a}_2$. This coding proved to be well adapted for encoding the boundary of discrete sets and is well known in discrete geometry as the *Freeman chain code* [2, 3]. It has been extensively used in many applications and allowed the design of elegant and efficient algorithms for describing geometric properties [4, 5, 7, 8, 9, 10].

3.1. Neighbor links

A discrete path consists of a list of points such that each point is at distance one from the previous one. This local property is useful for an efficient representation of sparse subsets of points in the radix tree. Indeed, it allows to process locally *most of the time* the nodes in a radix tree so that analyzing a path of length n requires finally a number of steps in $O(n)$. For this purpose, we superpose on the multidimensional radix tree $RT = (N, R)$ the *neighbor relation* represented by a second set of edges T in order to form the graph $\mathcal{G} = (N, R, T)$.

Definition 1. *Given two points $x, y \in \mathbb{Z}^d$, we say that x is a* neighbor *of y if* $||x - y||_\infty = 1$.

If x and y are neighbors then all of their coordinates are equal except for one which differs by exactly one. So, there exists a unique letter $\epsilon \in \Sigma$ such that $x + \overrightarrow{\epsilon} = y$, and in this case, we say that y is the *ϵ-neighbor* of x. Consequently, edges in T are labeled accordingly by setting $\textcircled{x} \overset{\epsilon}{\dashrightarrow} \textcircled{y}$ if y is the ϵ-neighbor of x. The edges in T are called *neighbor links*. For instance, in Figure 1, the neighbor links starting from the node $(2, 1)$ are distinguished with dashed arrows.

4. The algorithm

Adding 1 to an integer $\boldsymbol{x} \in \mathbb{B}^k$ is easily performed by a sequential function. Indeed, every positive integer can be written $\boldsymbol{x} = u1^i0^j$, where $i \geq 1$, $j \geq 0$, with $u \in \{\varepsilon\} \cup \left\{\mathbb{B}^{k-i-j-1} \cdot 0\right\}$. In other words, 1^i is the last run of 1's. The piece of code for adding 1 to an integer written in base 2 is

Algorithm 2: `addOne`

```
Input: x = u1^i 0^j
if ɹ ≠ 0 then
    return u1^i 0^{j-1} 1;
else if u = ε then
    return 10^i;
else
    return u0^{-1}10^i;
```

where 0^{-1} means to erase a 0. Clearly, the computation time of this algorithm is proportional to the length of the last run of 1's. Much better is achieved with the radix tree structure. Consider a d-tuple $x = (x_1, x_2, \ldots, x_d)$ and suppose you want the d-tuple $y = (x_1, \ldots, x_{i-1}, x_i + 1, x_{i+1}, \ldots x_d)$. One may use the above algorithm in order to increment the i-th coordinate of x, but having access to the node $\textcircled{x}$ of $\mathcal{G}$, there is a better way by simply following the neighbor link labeled a_i starting from $\textcircled{x}$ in order to reach $\textcircled{y}$. Now, suppose the graph $\mathcal{G} = (N, R, T)$ is incomplete, that is, (N, R) still respects the radix tree structure but N is some finite set of nodes and some edges of T are missing. One could still access $\textcircled{y}$ from $\textcircled{x}$ by going up in the radix tree structure until some ancestor of $\textcircled{y}$ is reached and then go down to $\textcircled{y}$.

Given a node $\textcircled{z} \in \mathcal{G}$, its *father* is denoted $f(\textcircled{z})$ or, by abuse of notation, we may simply write $f(z)$. The following technical lemma computes the father of a node's neighbor by using the fact that two neighbor nodes either share the same father, or their parents are neighbors. But, first, we introduce the convention that the root $\textcircled{r}$ of $\mathcal{G} = (N, R, T)$, which represents the point $\overrightarrow{0} = (0, 0, \ldots, 0)$ is its own father. Indeed, in Section 2 we saw that the radix tree structure imposes that if $\textcircled{r}$ has a son such that the edge of R leading to it has label $(0, 0, \ldots, 0)$ then this son would also represent the point $(0, 0, \ldots, 0)$. So, there is no contradiction to assume that $\textcircled{r}$ is its own son with label $(0, 0, \ldots, 0)$.

Lemma 1. *Let $\textcircled{x} \in N, \epsilon \in \Sigma$. Then*

$$f(x + \epsilon) = \begin{cases} f(x) \text{ if } \epsilon = a_i \text{ and } x_i \text{ is even, or } \epsilon = \overline{a}_i \text{ and } x_i \text{ is odd,} \\ f(x) + \overrightarrow{\epsilon} \text{ otherwise.} \end{cases}$$

Proof. It suffices to see that, for any $\epsilon \in \Sigma$, the translation of x by $\overrightarrow{\epsilon}$ affects only the i-th coordinate of x leaving all the other ones unchanged. Then, the

operation performed on the i-th coordinate is a transcription of the following trivial equality, for any integer z :

$$sz+1=\begin{cases} 2\frac{z}{2}+1 \text{ if } z \text{ is even,} \\ 2\left(\left\lfloor\frac{z}{2}\right\rfloor+1\right) \text{ if } z \text{ is odd.} \end{cases}$$

■

Figure 3 illustrates this process in dimension two where the nodes $(10110,\bullet)$ and $(10111,\bullet)$ share the same father while fathers of neighbor nodes $(\bullet,01011)$ and $(\bullet,01011)$ are distinct but share the same neighbor relation.

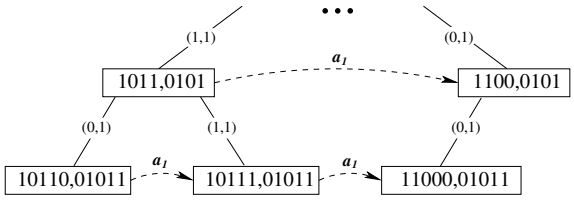

Figure 3: Computation of the father of a neighbor by using Lemma 1

Now, assume that the node $\textcircled{x}$ exists and that its ϵ-neighbor $\textcircled{y}$ does not. The translation $x+\overrightarrow{\epsilon}$ is obtained in three steps by the following rules:

1. take the edge in R to $f(\textcircled{x})$ and let α be the label of this edge;
2. take (or create) the edge in T from $f(\textcircled{x})$ to $\textcircled{z}=f(x)+\overrightarrow{\epsilon}$;
3. take (or create) the edge in R with label

$$\alpha'=(\alpha_1,\ldots,\alpha_{i-1},1-\alpha_i,\alpha_{i+1},\ldots,\alpha_d) \tag{3}$$

from $\textcircled{z}$ to $\textcircled{y}$.

By Lemma 1, we have $\boldsymbol{z}\cdot\alpha'=\boldsymbol{y}$, so that all it remains to do is to add the neighbor link $\textcircled{x}\overset{\epsilon}{\dashrightarrow}\textcircled{y}$. Then, a nonempty word $w\in\Sigma^n$ is sequentially processed to build the graph $\mathcal{G}_w$.

Starting from the empty graph $\mathcal{G}_\varepsilon$, the algorithm `readWord` sequentially reads $w\in\Sigma^*$, builds dynamically the graph $\mathcal{G}_w$ marking the corresponding node as visited, and determines if the path coded by w is self-intersecting, i.e. if some node is visited at least twice. We recall that the empty graph $\mathcal{G}_\varepsilon=(N_\varepsilon,R_\varepsilon,T_\varepsilon)$ is such that N_ε contains only the root, R_ε contains only the edge from the root to itself with label $(0,0,\ldots,0)$ and T_ε is empty.

Algorithm 3: `readWord`

Input: $w\in\{a,b,\overline{a},\overline{b}\}^*$
$\mathcal{G}\leftarrow\mathcal{G}_\varepsilon$;
$\textcircled{c}\leftarrow$ root of $\mathcal{G}$;
for $i\leftarrow 1$ **to** $|w|$ **do**
  $\epsilon\leftarrow w_i$;
  $\textcircled{z}\leftarrow$ `findNeighbor`$(\mathcal{G},\textcircled{c},\epsilon)$;
  if $\textcircled{z}$ *is* visited **then**
    w is self-intersecting.
  Mark $\textcircled{z}$ as *visited* ;
  $\textcircled{c}\leftarrow\textcircled{z}$;
w is not self-intersecting.

Algorithm 4: `findNeighbor`

Input: $\mathcal{G}=(N,R,T)$; $\textcircled{c}\in N$; $\epsilon\in\{a,b,\overline{a},\overline{b}\}$;
if *the link* $\textcircled{c}\overset{\boldsymbol{\epsilon}}{\dashrightarrow}\textcircled{z}$ *does not exist* **then**
  $\textcircled{p}\leftarrow f(\textcircled{c})$;
  if $f(\textcircled{c}+\boldsymbol{\epsilon})=\textcircled{p}$ **then**
    $\textcircled{r}\leftarrow\textcircled{p}$;
  else
    $\textcircled{r}\leftarrow$ `findNeighbor`$(\mathcal{G},\textcircled{p},\epsilon)$;
  $\textcircled{z}\leftarrow$ son of $\textcircled{r}$ corresponding to $\textcircled{c}+\boldsymbol{\epsilon}$;
  Add the neighbor link $\textcircled{c}\overset{\boldsymbol{\epsilon}}{\dashrightarrow}\textcircled{z}$;
return $\textcircled{z}$

The algorithm `findNeighbor` finds, and creates if necessary, the ϵ-neighbor of a given node. Thanks to Lemma 1, testing the condition on line 3 is performed in constant time. Indeed, this condition is tested by looking at the least significant bit of the binary expansion of the appropriate coordinate, say the i-th one, of the point c. This bit is given by the i-th bit of the label α, that is, the label of the edge in R linking $\textcircled{c}$ to its father. At line 6, if the node $\textcircled{z}$ does not exist, it is created. Note that the label α' of the edge going from $\textcircled{r}$ to its son $\textcircled{z}$ is computed in a single step from α as shown in Equation 3.

Clearly, the time complexity of this algorithm is entirely determined by the recursive call on line 5 since all other operations are performed in constant time assuming that d is fixed. Finally, note that after each call to `findNeighbor`, there always exist a neighbor link $\textcircled{c} \xrightarrow{\epsilon} \textcircled{z}$.

5. Complexity analysis

The goal of this section is to establish the overall lime linearity of our algorithm to detect path intersections. In order to do so, we assume that the dimension d is fixed, so that 2^d is considered as a constant.

The key point in the analysis rests on the fact that each recursive call in Algorithm 4 leads to the addition of a neighbor link and that a recursive call is performed on a node only when looking for one of its sons' neighbors. This implies that given a node $\textcircled{z} \in N$, when all the neighbor links starting from one of its children and leading to a node that is not one of its children have been added, there will never be another recursive call reaching $\textcircled{z}$. Since a node has at most 2^d sons and each of these sons has at most d neighbors not sharing the same father, the number of recursive calls on a single node is bounded by $d2^d$. It remains to show that the number of nodes in the graph is proportional to the length of the path analyzed.

First, consider the nodes marked as *visited* after having read the word w. For each letter read, exactly one node is marked as *visited*, so that their number is $|w|$. In order to bound the number of *non-visited* nodes, we need a technical lemma.

We define the function $f : N \to N$ that associates to a node its father in the tree $\mathcal{G}$ again with the convention that the root is its own father. This function extends to sets of nodes in the usual way: for $M \subseteq N$, the fathers of M are $f(M) = \{f(\textcircled{s}) \mid \textcircled{s} \in M\}$; moreover, f can be iterated to get the subset $f^h(M)$ of its *ancestors* of rank h. Clearly, f is a contraction since $|f(M)| \leq |M|$, and there is a unique ancestor of all nodes, namely the root.

Lemma 2. *Let $M = \{n_1, n_2, \ldots, n_k\} \subset N$ be a set of five nodes such that $(n_i, n_{i+1}) \in T$ for $i = 1, 2, \ldots, k$. Then, $k = 2^d + 1$ implies $|f(M)| \leq 2^d$.*

Proof. Any point $x \in \mathbb{Z}^d$ has $2d$ neighbors among which exactly d share the same father as x. Consider the function χ that associate to a point x the set of its neighbors that do not share the same father. By iteration of the function χ on a point x one obtains a stable set of cardinality 2^d.

It follows that for any $i \in \{1, 2, \ldots, k-1\}$, either x_i and x_{i+1} share the same father, or $x_{i+1} \in \chi(x_i)$ so that $|M| = 2^d + 1$ implies $|f(M)| \leq 2^d$. ∎

Using this lemma, we can now bound the total number of nodes inserted in the tree $\mathcal{G}_w$ by a factor which is linear with respect to $|w|$.

Lemma 3. *Given a word $w \in \Sigma^n$ and the graph $\mathcal{G}_w = (N, R, T)$, the number of nodes in N is in $\mathcal{O}(n)$.*

Proof. Let $N_v \subseteq N$ be the set of visited nodes, and h be the height of the tree (N, R). It is clear that $N = \bigcup_{0 \leq i \leq h} f^i(N_v)$, and so

$$|N| \leq \sum_{0 \leq i \leq h} |f^i(N_v)|. \tag{4}$$

By construction, the set N_v forms a sequence of nodes such that two consecutive ones are neighbors since they correspond to the path coded by w. Thus, by splitting this sequence of nodes in blocks of length $2^d + 1$, the previous lemma applies, and we have

$$|f(N_v)| \leq 2^d \left\lceil \frac{|N_v|}{2^d + 1} \right\rceil \leq \gamma(|N_v| + 2^d), \tag{5}$$

where $\gamma = \frac{2^d}{2^d+1}$. By Lemma 1, two neighbors either share the same father or have different fathers that are neighbors, and so it is for the sets $f(N_v), f^2(N_v), \ldots, f^h(N_v)$. Hence, by combining inequalities (4) and (5), we obtain the following bound

$$\begin{aligned} |N| &\leq \sum_{0 \leq i \leq h} \left|f^i(N_v)\right| \leq \sum_{0 \leq i \leq h} \left(\gamma^i |N_v| + \sum_{0 \leq j \leq i} \gamma^j 2^d \right) \\ &< |N_v| \left(\frac{1}{1-\gamma} \right) + 2^d \sum_{0 \leq i \leq h} \left(\frac{1}{1-\gamma} \right) \leq (2^d + 1)|N_v| + 2^d(2^d + 1)h. \end{aligned}$$

Since the height h of the tree (N, R) is exactly the number of bits needed to write the coordinates of the nodes in N, $h \in \mathcal{O}(\log n)$ which provides the claimed bound. ∎

6. Removing the assumption

In order to remove the assumption that starting from the origin the path analyzed always go through points with non-negative coordinates, several techniques may be used to handle paths having negative coordinates. For instance, since the property of being self intersecting or not is invariant by translation, it suffices to translate the path conveniently. This can be achieved by making one pass on the word w to determine the starting node $\textcircled{s}$ as follows:

(a) $\textcircled{s} \leftarrow (n, n, \ldots, n)$, where $n = |w|$;
(b) $\textcircled{s} \leftarrow (x_1, x_2, \ldots, x_d)$ where each x_i is determined from the extremal value of the i-th coordinate.

In both cases it takes $\mathcal{O}(n)$ time for reading the word and $\mathcal{O}(\log n)$ time and space to represent $\textcircled{s}$ in the radix tree by using Algorithm 1. Then the path is encoded in the radix tree starting from $\textcircled{s}$.

This solution is not completely satisfactory since it requires a linear preprocessing time and making it unsuitable in the case of streaming data where no assumption on the size or structure of the data holds. A second and better solution consists in representing the whole discrete space $\mathbb{Z}^d$ by combining 2^d radix tree structures, one for each hyper-octant (one for each of the four quadrant if $d = 2$, one for each of the eight octants if $d = 3,\dots$). To do so, we define an alternative initial graph $\mathcal{G}_\varepsilon^\pm = (N_\varepsilon^\pm, R_\varepsilon^\pm, T_\varepsilon^\pm)$ as follows:

- $N_\varepsilon^\pm = \{r_i \,|\, i \in \{0, 1, \dots, 2^d\}\}$. If $b_1 b_2 \dots b_d$ be the binary expansion of i on d bits, then r_i represents the point $(-b_1, -b_2, \dots, -b_d)$ and is the root of the tree representing the points of the same hyper-octant.
- $R_\varepsilon^\pm$ contains exactly one edge going from each root r_i to itself such that the label of the edge is given by the binary expansion of i on d bits.
- $T_\varepsilon^\pm$ contains all the possible neighbor links between the node of $N_\varepsilon^\pm$, that is, exactly d links starting on each node.

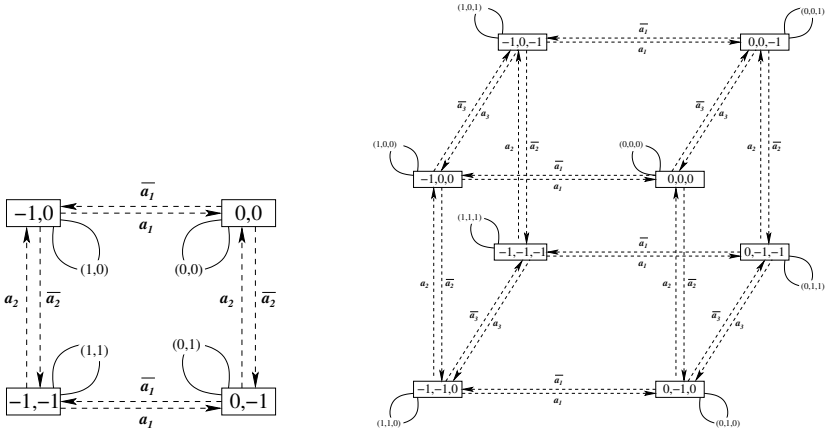

Figure 4: The graph $\mathcal{G}_\varepsilon^\pm$ for the cases $d = 2$ (left) and $d = 3$ (right).

Figure 4 illustrates the initial graph $\mathcal{G}_\varepsilon^\pm$ for the cases $d = 2$ and $d = 3$. See also Figure 5 for an illustration of the first two levels of the complete graph $\mathcal{G}^\pm$ in the case $d = 2$.

Note that, as before, each root is initialized as its own father with a labeling consistent with Equation 2 since $0 = 2(0) + 0$ and $-1 = 2(-1) + 1$. This labeling amounts to using the well known *two's-complement* arithmetic system commonly used to represent negative numbers in computer systems.

The two's complement notation ensures that Equation (2) holds for any positive of negative coordinates. Finally, since all roots are linked together by neighbor links, going from a radix tree to another is performed using recursion in a completely transparent manner.

Theorem 1. *Given a word $w \in \Sigma^n$, Algorithm 3 tests if the path coded by w intersects itself with a time and space complexity in $\mathcal{O}(n)$.*

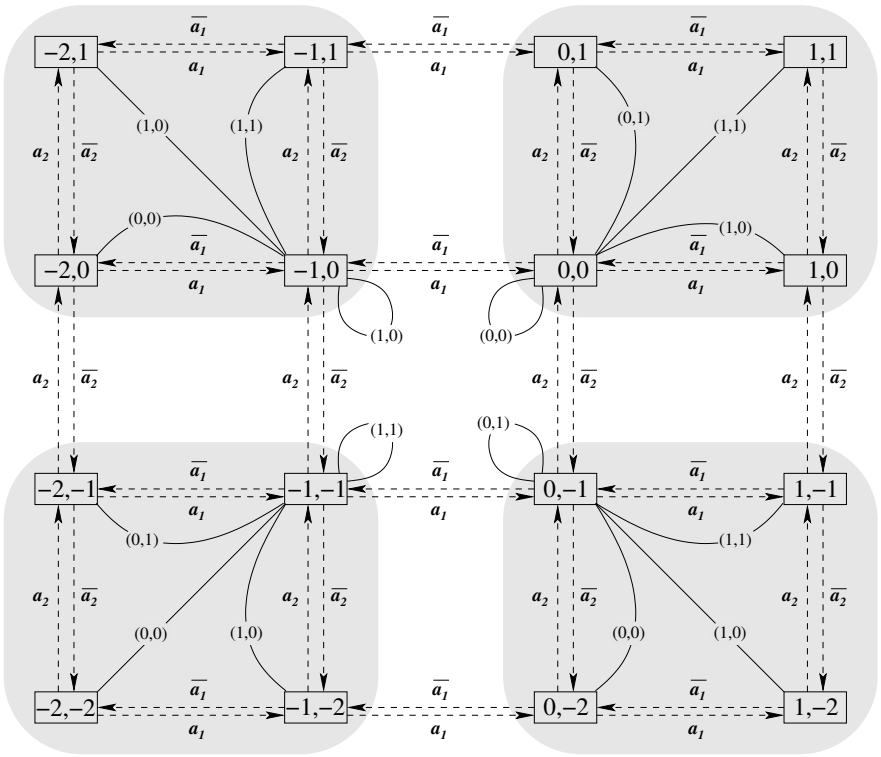

Figure 5: The first two levels of the complete graph.

Proof. Using the two's complement notation, Lemma 1 still applies to nodes with negative coordinates. Going from a tree to another is allowed by the fact that neighbor links between the roots of each subgraph are added in the initialization phase (see Figure 4). Indeed, when a quadrant (octant or hyper-octant) change occurs, recursion is used on extremal nodes of the subgraph (these are the nodes for which at least one neighbor is in another tree) until a neighbor link leading to the appropriate tree is reached. The first time this process occurs, no such neighbor link will only be found once the root of the tree is reached and a link leading its neighbor has been built during the initialization phase. Note that the whole initialization of the graph $\mathcal{G}_\varepsilon^\pm$ requires the creation of 2^d nodes and $d2^d$ neighbor links.

Finally, Lemma 3 bounds the number of nodes by some linear factor providing the claimed space complexity. On the other hand, time complexity is also bounded since, as mentioned previously, the total computation time is less than $d2^d$ times the number of nodes. This is due to the fact that once all sons of a node are linked to their neighbors, there might never be any recursive call reaching this node. ∎

A careful analysis of the arguments presented here for establishing the time complexity reveals a linearity constant quite large, especially in high dimensional cases. Indeed, we provided two bounds, a first one for the number of recursive calls on a single node that is $d2^d$, and a second one for the overall number of

nodes that is $2^d n$ where n is the length of the path.

A simple multiplication provides an upper bound that is enough to show the linearity of our algorithm but, as illustrated in Section 7.1 the result is far from being tight. Not only are both of these bounds unachievable but clearly the paths that would cause a high number of recursive calls on the same nodes are not the paths that would cause a high number of nodes, see for instance Figures 6 for an example of a path causing $d2^d$ recursive calls on the same node and Figure 7 for examples of paths causing the creation of a high number of nodes.

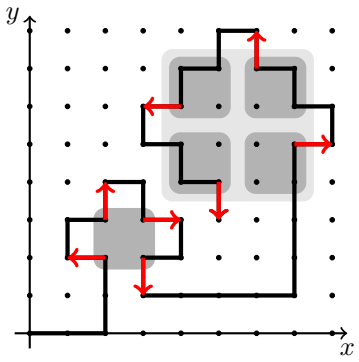

Figure 6: One of the shortest non-intersecting paths requiring $d2^d$ recursive calls reaching the node $(1, 1)$. The dark gray zones group the sons of a same node while the light gray zone groups the grand-sons of $(1, 1)$. The red arrows highlight the steps that cause recursive calls reaching $(1, 1)$.

7. Performance issues and comparison

Among the many ways of solving the intersection problem, the naive sparse matrix representation that requires an $O(n^d)$ space and initialization time is eliminated in the first round. When efficiency is concerned, there are two well-known approaches for solving it: one may store the coordinates of the visited points and sort them, then check whether two consecutive sets of coordinates are equal or not (we call this *sorting algorithm*). One may also store sets of coordinates in a self balanced-tree such as a red-black tree (RB-tree for short) and check for each new set of coordinates if it is already present or not (the *RB-tree algorithm*). Let us first assume that the path $w \in \Sigma^n$ is not self intersecting. Then the length of the largest coordinate is $O(\log n)$. But the largest coordinate is also $\Omega(\log n)$ because if the path is not self intersecting, the minimum coordinates are obtained when the points remain in a d-dimensional hyper-cube centered on the origin with $\sqrt[d]{n}$ side length. Since $\log(\sqrt[d]{n}) = \frac{1}{d} \log n$ the largest coordinate is also $\Omega(\log n)$. Thus the storage of the largest coordinate is in $\Theta(\log n)$ and the whole storage costs $\Theta(n \log n)$.

Sorting n points can be done in $\Theta(n \log n)$ swaps or comparisons. But each swap or comparison costs $\Theta(\log n)$ clock ticks. Then the whole computation time is $\Theta(n \log^2 n)$. In our algorithm, the storage cost is $\Theta(k)$ while computation time is bounded by $\Theta(k)$, where k is the index of the second occurrence of the point appearing at least twice in the path or the path length if it is not self

intersecting. Unlike the sorting algorithm, there is no need to store the whole path: the computation is performed dynamically.

With our algorithm, storing the necessary data costs, both on average and in worst case, $\mathcal{O}(k)$ if k can be stored in a machine word and $\mathcal{O}(k \log k)$ otherwise. Similarly for the time complexity. We summarize :

Algorithm	Unified Cost RAM model		General Case	
	Time	Space	Time	Space
Sorting	$n \log n$	n	$n \log^2 n$	$n \log n$
RB-tree	$k \log k$	k	$k \log^2 k$	$k \log k$
Our	k	k	$k \log k$	$k \log k$

Consider the simpler problem of checking if a path is closed, that is if for each $\epsilon \in \Sigma$ we have $|w|_\epsilon = |w|_{\bar{\epsilon}}$. The cost of storing the number $|w|_\epsilon$ of occurrences of each elementary step is in $\mathcal{O}(1)$ if each of these numbers can be stored in a single machine word, or in $\mathcal{O}(\log n_\epsilon)$ otherwise. Increasing or decreasing the number $|w|_\epsilon$ by 1 costs $\mathcal{O}(1)$ on average for n consecutive increment or n consecutive decrement. On the other hand, at worst case, a single increment or decrement may cost $O(\log n)$ and some paths realize this worst case frequently enough to produce an overall computation time in $\mathcal{O}(n \log n)$.

7.1. Numerical results

Our algorithm was implemented in C++ and tested on numerous examples. Numerical results presented here are all based on the analysis of the words $w_{(n,d)}$ that code a path self-intersecting on the last letter, defined as :

$$w_{(n,d)} = a_1^k \cdot a_2^k \cdots a_d^k \cdot \overline{a}_1^{2k} \cdot \overline{a}_2^{2k} \cdots \overline{a}_d^{2k} \cdot a_1^{2k} a_2^{2k} \cdots a_{d-1}^{2k} \cdot a_d^k$$

where $k = \left\lfloor \frac{n}{5d-1} \right\rfloor$ so that $|w_{(n,d)}| \sim n$. Figure 7 illustrates the *visited* and *non-visited* nodes when such a word is analyzed.

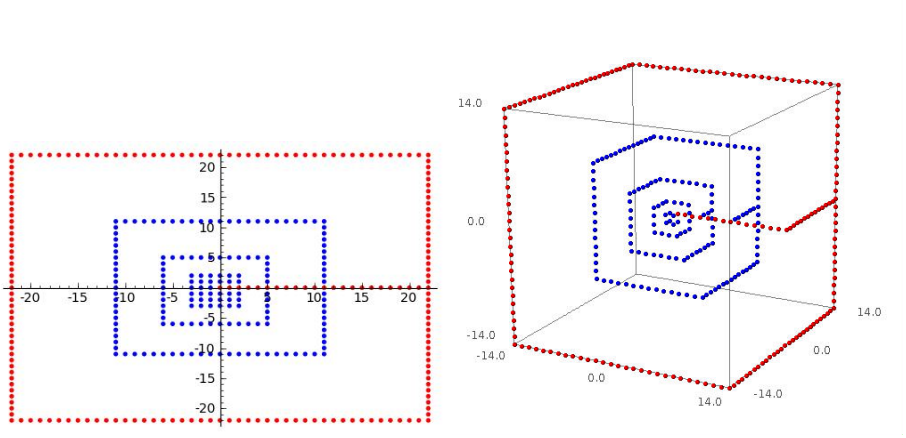

Figure 7: Representation of nodes created while analyzing the words $w_{(n,d)}$: $w_{(200,2)}$ (left) and $w_{(200,3)}$ (right), with *visited* nodes (red) and *non-visited* (blue).

Recall that each node of the radix-tree build may have up to $2d$ neighbors and 2^d sons, and that all links can be stored in arrays so that each node contains

an array of pointers of length $2d + 2^d$. Of course, this technique allows that once a node is reached, all its sons and neighbors are accessed in constant time. Nevertheless, as d grows, it requires a greater initialization cost for each node. In Figure 8 (left), this implementation with arrays is compared to an alternative implementation where the pointers in each node are stored by means of a self-balancing binary search tree (in this case the C++ standard library's `STL Map`). In an implementation with trees, the time cost is almost independent from the dimension d. Figure 8 (right) suggests that the extra time required as d grows seems to be mostly due to the initialization phase when the 2^d roots are linked together.

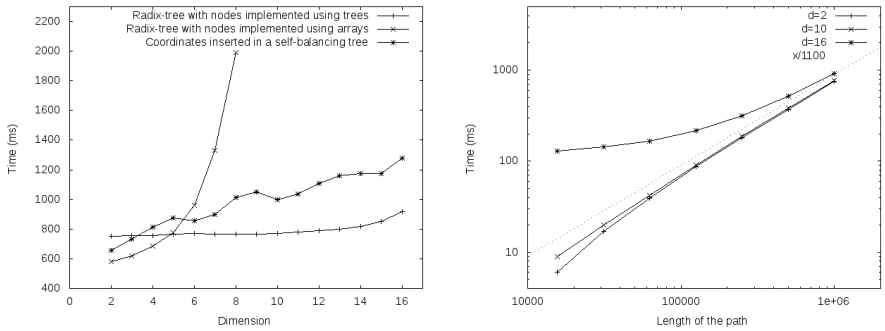

Figure 8: Timing results using the words $w_{(n,d)}$. Left: a comparison with the *RB-tree* algorithm; two different implementations of our data structure are compared, a first one where each node contains two self-balancing trees for indexing the neighbors and sons, a second one where the links to neighbors and sons are stored in arrays. Timings were obtained by testing the words $w_{(n,d)}$ with $n = 1000000$ and $d \in \{2, 3, \ldots, 16\}$. Right: computation times using our algorithm with node implemented using self-balanced trees. For both charts, each point is the average time of 20 computations.

The above results clearly show that the linearity constant $d2^{2d}$ is far from being reached in practice. We leave as an open problem the supply of a tight worst-case bound on the time linearity constant of this algorithm.

8. Concluding remarks

The first advantage of our algorithm is that in the case where nodes are implemented with arrays of pointers, ordering of edges can be used to avoid labeling of both nodes and edges. Moreover, the neighbor relation T as presented is not implemented in its symmetric form. It could be easily done since each time a neighbor link $\textcircled{c} \xrightarrow{\epsilon} \textcircled{z}$ is added at line 7 of Algorithm 4, we can add its symmetric link $\textcircled{z} \xrightarrow{\bar{\epsilon}} \textcircled{c}$ at constant cost. This does not change the overall complexity, and further analysis is required for determining if it is worthwhile. On the other hand, our algorithm is useful for solving a series of related problems in discrete geometry, with linear time and space complexity.

In dimension $d = 2$, determining if $w \in \Sigma^n$ is the Freeman chain code of a connected discrete figure. It suffices to check that the path is non-intersecting until the last letter is read. At this point, the starting point $\vec{0}$ must be reached. This does not penalize the linear algorithms for determining, for instance, if a discrete figure is digitally convex [8], or if it tiles the plane by translation [9].

Determining if a path w crosses itself. Again, in dimension 2, the fact that a discrete path visits twice the same point is by definition an intersection. However, one may define the more precise notion of crossing. For instance, the path coded by $a_1a_1a_2\overline{a}_1\overline{a}_2\overline{a}_2\overline{a}_1a_2$ is considered as crossing while points visited twice in the path $a_1a_1a_1a_2\overline{a}_1\overline{a}_2\overline{a}_1a_2\overline{a}_1\overline{a}_2$ designate a nill area zone and no crossing occurs. Deciding if self-intersecting path is crossing or not amounts to check local conditions, describing all the possible configurations (See [9] Section 4.1).

Node multiplicity. By replacing the "visited/unvisited" labeling of nodes with a counter (set to 0 when a node is created), the number of times a node is visited is computed by replacing the lines 6, 7 and 8 in Algorithm 3 by the incrementation of this counter. Then, the obsolete line 10 has to be removed.

Intersection of distinct paths. Given two distinct paths u and v of length bounded by $n = \max\{|u|, |v|\}$, with starting points x and y respectively, their intersection is computed by constructing first the graph $\mathcal{G}_u$ using Algorithm 3. Then the point $z = y - x$ in computed and the node $\textcircled{z}$ is added into $\mathcal{G}_u$ using Algorithm 1. Finally it suffices to use again Algorithm 3 starting from the node $\textcircled{z}$. Again the overall algorithm remains in $\mathcal{O}(n)$. As a byproduct of this construction, given two non-intersecting closed paths u and v, it is decidable whether the interior of u is included in the interior of v; and consequently one may compute the exterior envelope of discrete figures.

Acknowledgements. The authors wish to thank Julien Cassaigne for helpful discussions about this data structure during the Math Info 2010 "*Towards new interactions between mathematics and computer science*" conference in Marseille.

Note. A preliminary version (in French) of some of the results presented here appears in the doctoral thesis of Xavier Provençal [10], supported by a scholarship from FQRNT (Québec).

References

[1] S. Brlek, M. Koskas, X. Provençal, A linear time and space algorithm for detecting path intersection, in: S. Brlek, C. Reutenauer, X. Provençl (Eds.), Discrete Geometry for Computer Imagery, volume 5810 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2009, pp. 397–408.

[2] H. Freeman, On the encoding of arbitrary geometric configurations, IRE Trans. Electronic Computer 10 (1961) 260–268.

[3] H. Freeman, Boundary encoding and processing, in: B. Lipkin, A. Rosenfeld (Eds.), Picture Processing and Psychopictorics, Academic Press, New York, 1970, pp. 241–266.

[4] S. Brlek, G. Labelle, A. Lacasse, A note on a result of Daurat and Nivat, in: C. de Felice, A. Restivo (Eds.), Proc. DLT 2005, 9-th International Conference on Developments in Language Theory, number 3572 in LNCS, Springer-Verlag, Palermo, Italia, 2005, pp. 189–198.

[5] S. Brlek, G. Labelle, A. Lacasse, Properties of the contour path of discrete sets, Int. J. Found. Comput. Sci. 17 (2006) 543–556.

[6] I. Debled-Rennesson, J.-L. Rémy, J. Rouyer-Degli, Detection of the discrete convexity of polyominoes, Discrete Appl. Math. 125 (2003) 115–133.

[7] S. Brlek, J.-O. Lachaud, X. Provençal, Combinatorial view of digital convexity., in: D. Coeurjolly, I. Sivignon, L. Tougne, F. Dupont (Eds.), Discrete Geometry for Computer Imagery, 14th International Conference, DGCI 2008, Lyon, France, April 16-18, 2008, Proceedings, volume 4992 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 57–68.

[8] S. Brlek, J.-O. Lachaud, X. Provençal, C. Reutenauer, Lyndon+Christoffel = digitally convex, Pattern Recognition 42 (2009) 2239–2246.

[9] S. Brlek, X. Provençal, J.-M. Fédou, On the tiling by translation problem, Discr. Appl. Math. 157 (2009) 464–475.

[10] X. Provençal, Combinatoire des mots, géométrie discrète et pavages, Ph.D. thesis, D1715, Université du Québec à Montréal, 2008.

[11] R. Finkel, J. Bentley, Quad trees: A data structure for retrieval on composite keys, Acta Informatica 4(1) (1974) 1–9.

[12] D. E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, Reading, Massachusetts, 1998.

[13] M. Lothaire, Applied Combinatorics on Words, Cambridge University Press, Cambridge, 2005.