A linear time and space algorithm for detecting path intersection ☆

Srečko Brlek[a], Michel Koskas[b], Xavier Provençal[d,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.*
[d] *LIRMM, CNRS UMR 5506, Université Montpellier II, 161 rue Ada, 34095 Montpellier cedex 05.*

Abstract

For discrete sets coded by the Freeman chain describing their contour, several linear algorithms have been designed for determining their shape properties. Most of them are based on the assumption that the boundary word forms a closed and non-intersecting discrete curve. In this article, we provide a linear time and space algorithm for deciding whether a path on a square lattice intersects itself. forms the contour of a discrete figure. This is achieved by adding a radix tree structure over a quadtree, where nodes represents grid points, enriched with neighborhood links that are essential for obtaining linearity. Due to its simplicity, this algorithm has many applications and, as an illustrative example, we use it for determining efficiently a solution to the more general problem of multiple paths intersection.

Keywords: Freeman code, lattice paths, self-intersection, radix tree, discrete figures, data structure.

1. Introduction

Many problems in discrete geometry involve the analysis of the contour of discrete sets. A convenient way to represent them is to use the well-known Freeman chain code [1, 2] which encodes the contour by a word w on the four letter alphabet $\Sigma = \{a, b, \overline{a}, \overline{b}\}$, corresponding to the unit displacements in the four directions (right, up, left, down) on a square grid. Among the many problems that have been considered in the literature, we mention : computations of

☆with the support of NSERC (Canada)
Email addresses: Brlek.Srecko@uqam.ca (Srečko Brlek), michel.koskas@agroparistech.fr (Michel Koskas), provencal@lirmm.fr (Xavier Provençal)

statistics such as area, moment of inertia [3, 4], digital convexity [5, 6, 7], and tiling of the plane by translation [8, 9]. All of the above mentioned problems are solved by using algorithms that are linear in the length of the contour word, but often it is assumed that the path encoded by this word does not intersect itself. While it is easy to check that a word encodes a closed path (by checking that the word contains as many a as $\overline{a}$, and as many b as $\overline{b}$), checking that it does not intersect itself requires more work. The problem amounts to check if a grid point is visited twice. Of course, one might easily provide an $\mathcal{O}(n \log n)$ algorithm where sorting is involved, or use hash tables providing a linear time algorithm on average but not in worst case.

The goal of this paper is to remove this major drawback by providing a linear time and space algorithm in the worst case checking if a path encoded by a word visits any of the grid points twice. Section 2 provides the basic definitions and notation used in this paper. It also contains the description of the data structures used in our algorithm: it is based on a quadtree structure [10], used in a novel way for describing points in the plane, combined with a radix tree (see for instance [11]) structure for the labels. In Section 3 the algorithm is described in details. The time and space complexity of the algorithm is carried out in Section 4, followed by a discussion on complexity issues, with respect to the size of numbers and bit operations involved. In Section 5 we consider the problem of multiple paths intersection, and finally, we provide a list of possible applications where the algorithm is useful and essential.

2. Preliminaries

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 $\Sigma = \{a, b, \overline{a}, \overline{b}\}$.

To any word $w \in \Sigma^*$ is associated a vector $\overrightarrow{w}$ by the morphism $\overrightarrow{\cdot} : \Sigma^* \longrightarrow \mathbb{Z} \times \mathbb{Z}$ defined on the elementary translation $\overrightarrow{\epsilon} = (\epsilon_1, \epsilon_2)$ corresponding to each letter $\epsilon \in \Sigma$:

$$\overrightarrow{a} = (1, 0), \quad \overrightarrow{\overline{a}} = (-1, 0),$$
$$\overrightarrow{b} = (0, 1), \quad \overrightarrow{\overline{b}} = (0, -1),$$

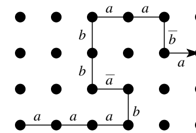


Figure 1: $w = aaab\overline{a}bbaa\overline{b}a$.

and such that $\overrightarrow{u \cdot v} = \overrightarrow{u} + \overrightarrow{v}$. The set of elementary translations allows to draw each word as a 4-connected path in the plane starting from the origin, going *right* for a letter a, *left* for a letter $\overline{a}$, *up* for a letter b and *down* for a letter $\overline{b}$ (Figure 1). This coding proved to be very convenient for encoding the boundary of discrete sets and is well known in discrete geometry as the *Freeman chain code* [1, 2].

It has been extensively used in many applications and allowed the design of elegant and efficient algorithms for describing geometric properties.

The underlying principle of our algorithm is to build a graph whose nodes represent points of the plane. For that purpose, the plane is partitioned as in Figure 2, where the point $(2,1)$ is outlined with its four sons (solid arrows) and its four neighbors (dashed arrows). The sons of a node are grouped in grey zones, while dashed lines separate the levels of the tree. Each node has two possible states : *visited* or *not visited*. New nodes are created while reading the word $w = w_1 w_2 \cdots w_n$ from left to right. For each letter w_i, the node corresponding

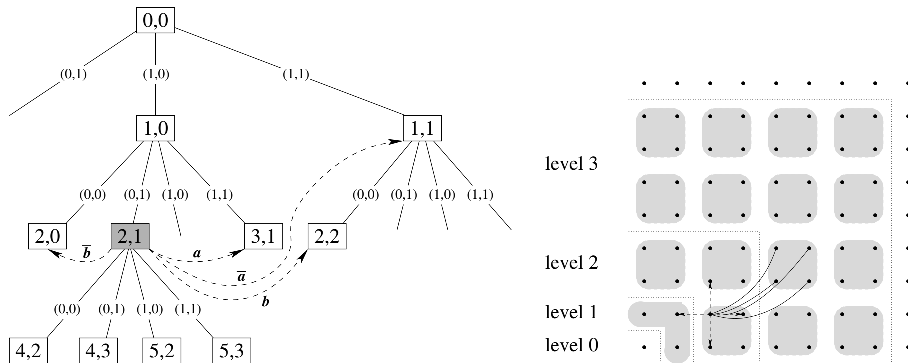


Figure 2: Left: the point $(2,1)$ with its sons and neighbors in the radix-tree. Right: the partition of $\mathbb{N} \times \mathbb{N}$ defined by the radix-tree.

to the point $\overrightarrow{w[1..i]}$ is written as *visited* and of course, if at some point a node is visited twice then the path is not self-avoiding and the algorithm stops. During the process is built a graph $\mathcal{G} = (N, R, T)$ where N is a set of nodes associated to points of the plane, R and T are two distincts sets of oriented edges. The edges in R give a quadtree structure on the nodes while the edges in T are links from each node to its *neighbors*, for which we give a precise definition.

Definition 1. *Given a point* $(x,y) \in \mathbb{Z}^2$*, we say that* (x',y') *is a* neighbor *of* (x,y) *if there exists* $\epsilon \in \Sigma$ *such that* $(x',y') = (x,y) + \boldsymbol{\epsilon} = (x+\epsilon_1, y+\epsilon_2)$.

When we want to discriminate the neighbors of a given point (x,y), for each $\epsilon \in \Sigma$, we say that (x',y') is an ϵ-*neighbor* of (x,y) if $(x',y') = (x,y) + \boldsymbol{\epsilon}$.

2.1. Data structure

First, we assume that the path is coded by a word w starting at the origin $(0,0)$, and stays in the first quadrant $\mathbb{N} \times \mathbb{N}$. This means that the coordinates of all points are nonnegative. Subsequently, this solution is modified in order to remove this assumption. Note that in $\mathbb{N} \times \mathbb{N}$, each point has exactly four neighbors with the exception of the origin $(0,0)$ which admits only two neighbors, namely $(0,1)$ and $(1,0)$, and the points on the half lines $(x,0)$ and $(0,y)$ with $x, y \geq 1$ which admit only three neighbors (see Figure 2).

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 [11, 12]), 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 $\mathbb{B}^* \times \mathbb{B}^*$ as follows.

A quadtree with a radix tree structure for points in the integer plane. As usual, the concatenation is extended to the cartesian product of words by setting for $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{B}^* \times \mathbb{B}^*$, and $(\alpha, \beta) \in \mathbb{B} \times \mathbb{B}$

$$(\boldsymbol{x}, \boldsymbol{y}) \cdot (\alpha, \beta) = (\boldsymbol{x} \cdot \alpha, \boldsymbol{y} \cdot \beta).$$

Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two binary words having same length. Then the rule

$$(\boldsymbol{x}, \boldsymbol{y}) \longrightarrow (\boldsymbol{x} \cdot 0, \boldsymbol{y} \cdot 0) + (\boldsymbol{x} \cdot 0, \boldsymbol{y} \cdot 1) + (\boldsymbol{x} \cdot 1, \boldsymbol{y} \cdot 0) + (\boldsymbol{x} \cdot 1, \boldsymbol{y} \cdot 1) \tag{1}$$

defines a $\mathcal{G}' = (N, R)$, sub-graph of $\mathcal{G} = (N, R, T)$, such that :

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

(ii) each node (except the root) has four sons;

(iii) if a node is labeled $(\boldsymbol{x}, \boldsymbol{y})$ then $|\boldsymbol{x}| = |\boldsymbol{y}|$;

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

By convention, edges leading to the sons are labeled by pairs from the ordered set $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$. These labels equip the quadtree with a *radix tree* structure for Equation (1) implies that (x', y') is a son of (x, y), if and only if

$$(x', y') = (2x + \alpha, 2y + \beta), \tag{2}$$

for some $(\alpha, \beta) \in \mathbb{B} \times \mathbb{B}$. Observe that any pair (x, y) of nonnegative integers is represented exactly once in this tree. Indeed, if $|\boldsymbol{x}| = |\boldsymbol{y}|$ (by filling with zeros at the left of the shortest one), the sequence of pairs of digits (the two digits in first place, the two digits in second place, and so on) gives the unique path in the tree leading to this pair. Of course the root may have up to three sons since no edge labeled $(0, 0)$ starts from the root.

neigbhor links. We superpose on G' the neigbhor relation given by the edges of T (dashed lines). More precisely, for each elementary translation $\epsilon \in \Sigma$, each node $\textcircled{z} = (x, y)$ is linked to its ϵ-neighbor $\textcircled{z} + \boldsymbol{\epsilon}$, when it exists. If a level k is fixed (see Figure 2), it is easy to construct the graph

$$\mathcal{G}^{(k)} = (N^{(k)}, R^{(k)}, T^{(k)})$$

such that

(i) if $(\boldsymbol{x},\boldsymbol{y}) \in N^{(k)}$, then $|\boldsymbol{x}| = |\boldsymbol{y}| = k$;

(ii) the functions $N^{(k)} \hookrightarrow \mathbb{N}\times\mathbb{N} \hookrightarrow \mathbb{B}^* \times \mathbb{B}^*$ are injective;

(iii) $R^{(k)}$ is the radix-tree representation : $(\mathbb{B}^{<k}\times\mathbb{B}^{<k})\times(\mathbb{B}\times\mathbb{B}) \xrightarrow{\bullet} \mathbb{B}^{\leq k}\times\mathbb{B}^{\leq k}$;

(iv) the neigbhor relation is $T^{(k)} \subseteq N \times (\mathbb{B}\times\mathbb{B}) \times N$.

Note that the labeling in Figure 2 is superfluous: each node represents indeed an integer unambiguously determined by the path from the root using edges in R; similarly for the ordered edges. Moreover, if a given subset $M \subset \mathbb{N}\times\mathbb{N}$ has to be represented, then one may trim the unnecessary nodes so that the corresponding graph $\mathcal{G}_M$ is not necessarily complete.

3. 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 1: `addOne`

```
Input: x = u1^i0^j
if j ≠ 0 then
  └ return u1^i0^{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, as wee shall see. Given a node ⓩ, its *father* is denoted f(ⓩ) and we write $f(x,y)$ or $f(\boldsymbol{x},\boldsymbol{y})$ if its label is (x,y). The following technical lemma is a direct adaptation to $\mathbb{B}^*\times\mathbb{B}^*$ of the addition above.

Lemma 1. *Let $G^{(k)}$ be the complete graph representing $\mathbb{B}^{\leq k}\times\mathbb{B}^{\leq k}$ for some $k \geq 1$, $\epsilon \in \Sigma$, and ⓩ $= (\boldsymbol{x},\ \boldsymbol{y})$ be a node of N^k. If one of the four conditions holds:*

$$\begin{array}{lllllll}(\text{i}) & \epsilon = a & \text{and} & \boldsymbol{x}[\text{k}] = 0, & (\text{ii}) & \epsilon = \overline{a} & \text{and} & \boldsymbol{x}[\text{k}] = 1,\\ (\text{iii}) & \epsilon = b & \text{and} & \boldsymbol{y}[\text{k}] = 0, & (\text{iv}) & \epsilon = \overline{b} & \text{and} & \boldsymbol{y}[\text{k}] = 1,\end{array}$$

then f(ⓩ) $= f$(ⓩ $+ \boldsymbol{\epsilon}$). Otherwise, f(ⓩ) $+ \boldsymbol{\epsilon} = f$(ⓩ $+ \boldsymbol{\epsilon}$).

The process is illustrated for case (i) in Figure 3 where the nodes $(10110,\bullet)$ and $(10111,\bullet)$ share the same father while fathers of neigbhor nodes $(\bullet,01011)$ and $(\bullet,01011)$ are distinct but share the same neigbhor relation.

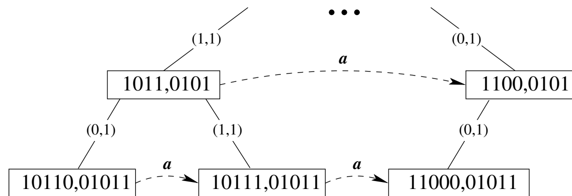


Figure 3: Computation of Lemma 1

Now, assume that the node $(\boldsymbol{x}, \boldsymbol{y})$ exists and that its neighbor $(x+1, y+0)$ does not. If $|\boldsymbol{x}| = |\boldsymbol{y}| = k$, then the translation $(x, y) + (1, 0)$ is obtained in three steps by the following rules:

1. take the edge in R to $f(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}[1..k-1], \boldsymbol{y}[1..k-1])$;
2. take (or create) the edge in T from $f(\boldsymbol{x}, \boldsymbol{y})$ to ⓩ $= f(\boldsymbol{x}, \boldsymbol{y}) + (1, 0)$;
3. take (or create) the edge in R from ⓩ to ⓩ $\cdot (0, \boldsymbol{y}[k])$.

By Lemma 1, we have ⓩ $\cdot (0, \boldsymbol{y}[k]) = (x+1, y+0)$, so that it remains to add the neigbhor link $(\boldsymbol{x}, \boldsymbol{y}) \overset{\boldsymbol{a}}{\dashrightarrow} (x+1, y+0)$. Then, a nonempty word $w \in \Sigma^n$ is sequentially processed to build the graph $\mathcal{G}_w$, and we illustrate the algorithm on the input word $w = aabb$.

- *Initialization*: the algorithm starts with the graph containing only the node $(0, 0)$ marked as visited. For convenience, the *non-visited* nodes $(0, 1), (1, 0)$, and the links from $(0, 0)$ to its neighbors are also added. This is justified by the fact that the algorithm applies to nonempty words. Since $(0, 0)$ is an ancestor of all nodes, this ensures that every node has an ancestor linked with its neighbors. The *current node* is set to $(0, 0)$ and this graph is called the *initial graph* $\mathcal{G}_\varepsilon$.

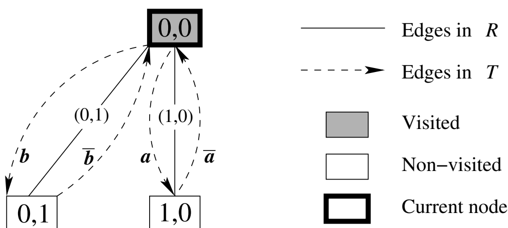
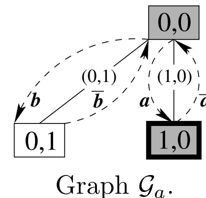


Figure 4: Initial graph $\mathcal{G}_\varepsilon$.

- *Read* $w_1 = a$: this corresponds to the translation $(0, 0) + (1, 0)$. A neigbhor link labeled $\boldsymbol{a}$ starting from $(0, 0)$ and leading to the node $(1, 0)$ does exist, so the only thing to do is to follow this link and mark the node $(1, 0)$ as visited. The current node is now set to $(1, 0)$, and this new graph is called $\mathcal{G}_a$.

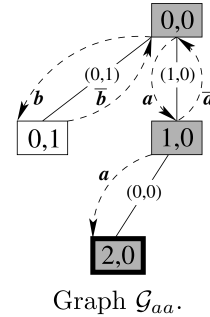


Graph $\mathcal{G}_a$.

• *Read* $w_2 = a$: this time, there is no edge in $\mathcal{G}_a$ labeled $\boldsymbol{a}$ starting from $(1,0)$. Using the translation rules above, we perform:

(1) go back to the father $f(1,0) = (0,0)$;
(2) follow the link $\boldsymbol{a}$ to $(1,0)$;
(3) add node $(2,0) \sim (1,0) \cdot (0,0) = (10,00)$.

Then an edge from $(1,0)$ to $(2,0)$ with label $\boldsymbol{a}$ is added to T. Finally the node $(2,0)$ is marked as *visited*, and becomes the current node.

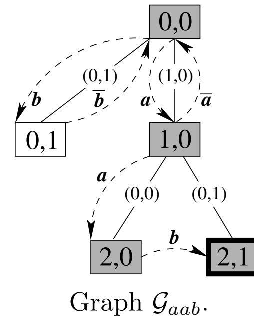


Graph $\mathcal{G}_{aa}$.

• *Read* $w_3 = b$: this amounts to perform the translation $(2,0)+(0,1)$. Since the edge to $f(2,0)$ is labeled by $(0,0)$, we know that the second coordinate of the current node $(2,0)$ is even. Therefore, $(2,1)$ and $(2,0)$ must be siblings, that is $f((2,0)+(0,1)) = f(2,0)$. What we need to do then is :

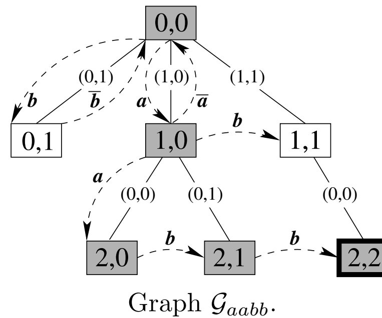
(1) go back to the father $f(2,0) = (1,0)$;
(2) follow the edge $\boldsymbol{b}$ if it exists.

Since it is does not exist, it must be created to reach the node $(2,1) \sim (10,01) = (1,0) \cdot (0,1)$. Again an edge from $(2,0)$ to $(2,1)$ with label $\boldsymbol{b}$ is added, $(2,1)$ is marked as *visited* and is now the current node.



Graph $\mathcal{G}_{aab}$.

• *Read* $w_4 = b$: since $f((2,1))$ has no neigbhor link labeled by $\boldsymbol{b}$, recursion is used to find (or build if necessary) the node corresponding to its translation by $\boldsymbol{b}$. This leads to the creation of the node $(1,1) \sim (0,0) \cdot (1,1)$ marked as *non-visited*. Then, the node $(2,2) \sim (1,1) \cdot (0,0)$ is added, marked as *visited*, and becomes the current node. Note that neigbhor links between $(1,0)$ and $(1,1)$, $(2,1)$ and $(2,2)$ are added in order to avoid searches.



Graph $\mathcal{G}_{aabb}$.

The algorithm `readWord` sequentially reads $w \in \Sigma^*$, builds dynamically the graph 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.

Algorithm 2: `readWord`

Input: $w \in \{a, b, \overline{a}, \overline{b}\}^*$

1. $\mathcal{G} \leftarrow \mathcal{G}_\varepsilon$;
2. $\textcircled{c} \leftarrow$ root of $\mathcal{G}$;
3. **for** $i \leftarrow 1$ **to** $|w|$ **do**
4. $\epsilon \leftarrow w_i$;
5. $\textcircled{z} \leftarrow$ `findNeighbor`$(\mathcal{G}, \textcircled{c}, \epsilon)$;
6.   **if** $\textcircled{z}$ *is* visited **then**
7. w is self-intersecting.
8.   **Mark** $\textcircled{z}$ as *visited* ;
9. $\textcircled{c} \leftarrow \textcircled{z}$;
10. w is not self-intersecting.

Algorithm 3: `findNeighbor`

Input: $\mathcal{G} = (N, R, T)$; $\textcircled{c} \in N$; $\epsilon \in \{a, b, \overline{a}, \overline{b}\}$;

1. **if** *the link* $\textcircled{c} \overset{\epsilon}{\dashrightarrow} \textcircled{z}$ *does not exist* **then**
2. $\textcircled{p} \leftarrow f(\textcircled{c})$;
3.   **if** $f(\textcircled{c} + \boldsymbol{\epsilon}) = \textcircled{p}$ **then**
4. $\textcircled{r} \leftarrow \textcircled{p}$;
   else
5. $\textcircled{r} \leftarrow$ `findNeighbor`$(\mathcal{G}, \textcircled{p}, \epsilon)$;
6. $\textcircled{z} \leftarrow$ son of $\textcircled{r}$ corresponding to $\textcircled{c} + \boldsymbol{\epsilon}$;
7.   Add the neigbhor link $\textcircled{c} \overset{\epsilon}{\dashrightarrow} \textcircled{z}$;
8. **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. At line 6, if the node $\textcircled{z}$ does not exist, it is created. Clearly, the time complexity of this algorithm is entirely determined by the recursive call on line 6 since all other operations are performed in constant time. Finally, note that after each call to `findNeighbor` on line 5 in Algorithm 2, there always exist a neigbhor link $\textcircled{c} \overset{\epsilon}{\dashrightarrow} \textcircled{z}$.

4. Complexity analysis

The key for analyzing the complexity of this algorithm rests on the fact that each recursive call in Algorithm 3 requires the addition of a neigbhor link and that a recursive call is performed on a node only when looking for one of its sons' neighbor. This implies that given a node $\textcircled{z} \in N$, when all neigbhor links of its children have been added, there will never be another recursive call on $\textcircled{z}$. Since a node has at most 4 sons and each of these sons has at most 2 neighbors not sharing the same father, the number of recursive calls on a single node is bounded by 8. It remains to show that the number of nodes in the graph is proportional to $|w|$.

First, consider the *visited* nodes. 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. Recall that the father function $f : N \setminus \{(0,0)\} \longrightarrow N$ extends to subsets of nodes in the usual way: for $M \subseteq N$, the fathers of M are $f(M) = \{f(\textcircled{s}) \,|\, \textcircled{s} \in M\}$. Moreover, f can be iterated to get $f^h(M)$, the *ancestors* of rank h of a subset M. 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* $M = \{n_1, n_2, n_3, n_4, n_5\} \subset N$ *a set of five nodes such that* $(n_i, n_{i+1}) \in T$ *for* $i = 1, 2, 3, 4$*, then,* $|f(M)| \leq 4$.

Proof. As shown in Figure 2, the nodes sharing the same father split the plane in 2×2 squares. As a consequence, at least two of the nodes n_1, n_2, n_3, n_4, n_5 must share the same father, providing the bound $|f(M)| \leq 4$. ∎

This allows to bound the number of nodes using the fact that all non-visited nodes are ancestors of visited ones: the only exception is the initialization step where the non-visited nodes $(0,1)$ and $(1,0)$ are created as leaves.

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

$$|N| \leq \sum_{0 \leq i \leq h} |f^i(N_v)|. \tag{3}$$

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 in blocks of length 5, the previous lemma applies, and we have

$$|f(N_v)| \leq 4 \left\lceil \frac{|N_v|}{5} \right\rceil \leq \frac{4}{5}(|N_v| + 4). \tag{4}$$

By Lemma 1, two neighbors either share the same father or have different fathers that are neighbors, so it is for the sets $f(N_v), f^2(N_v), \ldots, f^h(N_v)$. Hense, by combining inequalities (3) and (4), we obtain

$$\begin{aligned} |N| \leq \sum_{0 \leq i \leq h} \left| f^i(N_v) \right| &\leq \sum_{0 \leq i \leq h} \left(\left(\frac{4}{5}\right)^i |N_v| + \sum_{0 \leq j \leq i} \left(\frac{4}{5}\right)^j 4 \right) \\ &\leq |N_v| \left(\frac{1}{1 - \frac{4}{5}} \right) + 4 \sum_{0 \leq i \leq h} \left(\frac{1}{1 - \frac{4}{5}} \right) \leq 5|N_v| + 20h. \end{aligned}$$

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, $h \in \mathcal{O}(\log n)$ and thus $|N| \in \mathcal{O}(n)$. ∎

Note that the linearity constant obtained here is very large. Indeed, our goal here is to prove the linearity of the global algorithm, and not to provide a tight bound. With a more detailed analysis, the bound $|N| \leq 3|N_v| + 6h$ can be obtained for the number of nodes [13].

5. Arbitrary paths in all four quadrants

In order to deal with paths running in all four quadrants, one may use some technique to deal with 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 ⓢ as follows:

(a) ⓢ $\leftarrow (n, n)$, where $n = |w|$;
(b) ⓢ $\leftarrow (x, y)$ where x and y are determined from the extremal values.

In both cases, it takes $\mathcal{O}(n)$ steps to read the word and $\mathcal{O}(\log n)$ time and space to represent Ⓢ. Then the path is encoded in the radix-tree starting from Ⓢ. This is simply done by creating all the ancestors of Ⓒ before the beginning of the algorithm. This step requires an additional $O(\log(n))$ time and space preprocessing since the node (x, y) has exactly $\max\left(\lfloor\log_2(x)\rfloor, \lfloor\log_2(y)\rfloor\right)$ ancestors. This solution is not satisfactory since it requires a linear preprocessing time and making it non-suitable 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 not only the first quadrant but the whole discrete plane $\mathbb{Z}\times\mathbb{Z}$. To do so, we define an alternative initial graph $\mathcal{G}_\varepsilon^{(4)}$ illustrated in Figure 5. First, observe that the *initial graph* in Figure

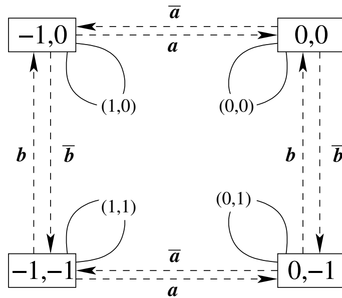


Figure 5: Graph $\mathcal{G}_\varepsilon^{(4)}$ representing all four quadrants at once.

4 does not need to be initialized with three nodes. Indeed, it is enough to start with the root $(0, 0)$ initialized as its own father. We call this graph $\mathcal{G}'_\varepsilon$. In this graph, the edge going from $(0, 0)$ to itself is labeled $(0, 0)$ so that the radix-tree structure relation given in Equation (2) (Section 2) is satisfied:

$$(0, 0) = (2 * 0 + 0, 2 * 0 + 0).$$

Now, using this single node, Algorithm 3 may be used to find the neighbors of $(0, 0)$. For instance, consider a call to `findNeighbor`$(\mathcal{G}'_\varepsilon, (0, 0), a)$. Since the binary writing of both x and y coordinates of Ⓒ $= (0, 0)$ ends with 0, according to Lemma 1 we have

$$f\left(Ⓒ + a\right) = f\left(Ⓒ\right).$$

On line 4, Ⓡ $= (0, 0)$ and on line 6, the node $(1, 0)$ is created as the son of $(0, 0)$ with an edge labeled $(1, 0)$. On line 7 the neigbhor link labeled a is added from $(0, 0)$ to $(1, 0)$.

The same idea is used to represent any quadrant of the discrete plane $\mathbb{Z}\times\mathbb{Z}$. It suffices to initialize a *root* per quadrant and use the two's complement for the binary representation of the negative coordinates. Using this notation, the representation of a negative number may start with an arbitrary number of ones on the left, as positive numbers may be written with an arbitrary number of zeros on the left. Consequently, each of these four roots is set as its own father with the following labeling:

- the root $(0,0)$ is its own father with label $(0,0)$,
- the root $(-1,0)$ is its own father with label $(1,0)$,
- the root $(0,-1)$ is its own father with label $(0,1)$,
- the root $(-1,-1)$ is its own father with label $(1,1)$,

The two's complement notation ensures that Equation (2) still holds. For instance, consider the four sons of node $(-1,0)$:

Edge's label	Son's label
$(0,0)$	$(-2,0) = (2*-1+0, 2*0+0)$
$(1,0)$	$(-1,0) = (2*-1+1, 2*0+0)$
$(0,1)$	$(-2,1) = (2*-1+0, 2*0+1)$
$(1,1)$	$(-1,1) = (2*-1+1, 2*0+1)$

Finally, in order to allow the passage between quadrants, it suffices to add all neigbhor links between the four roots as shown in Figure 5 and Figure 6.

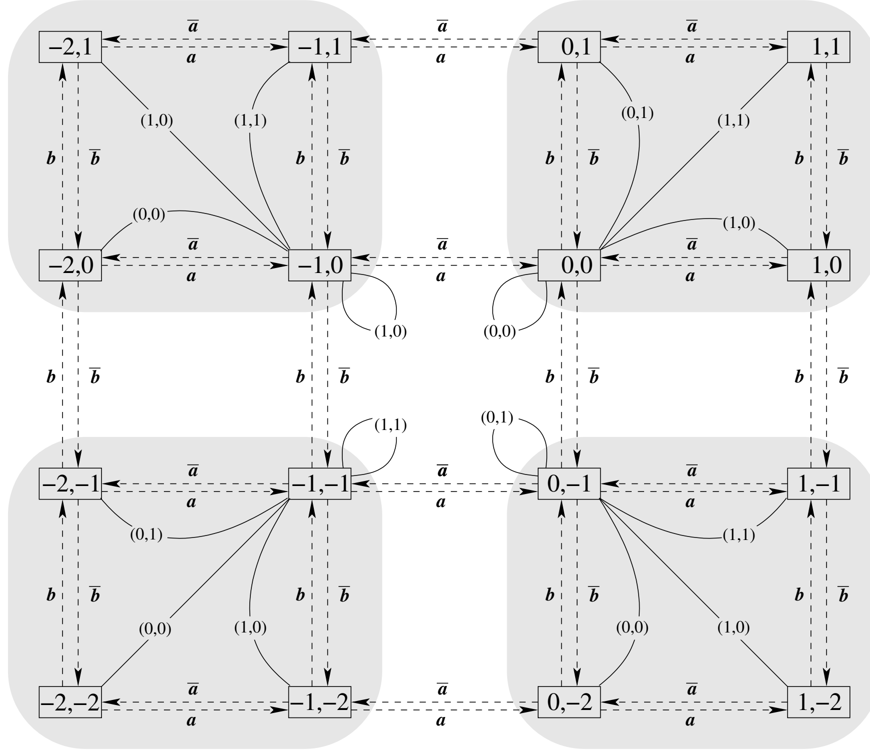


Figure 6: The first two levels of the complete graph.

Theorem 1. *Given a word $w \in \Sigma^n$, Algorithm 2 tests if the path coded by w intersects itself with a time and space complexity of $\mathcal{O}(n)$.*

Proof. Using the two's complement notation, Lemma 1 still applies to nodes with negative coordinates. Quadrant changes are ensured by the fact that neigbhor links between roots of each subgraph are added in the initialization phase (see Figure 5). Indeed, when a quadrant change occurs, recursion is used on extremal nodes of the subgraph (nodes with only three neighbors in the same radix-tree) until a neigbhor link leading to the other subgraph is reached. The first time this process occurs, no such neigbhor link will be found until the root of the subgraph is reached and a link leads to the appropriate subgraph's root.

Finally, Lemma 3 also applies which proves the linearity both in time and space of the overall algorithm. ∎

Performance issues and comparison. Among the many ways of solving the intersection problem, the naive sparse matrix representation that requires an $O(n^2)$ 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 if two consecutive sets of coordinates are equal or not (we call this *sorting algorithm*). One may also store the sets of coordinates in an AVL-tree and check for each new set of coordinates if it is already present or not (the *AVL 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 square centered on $(0, 0)$ with $\sqrt{n}$ side length. Since $\log(\sqrt{n}) = \frac{1}{2}\log n$ 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 $\Theta(n \log n)$.

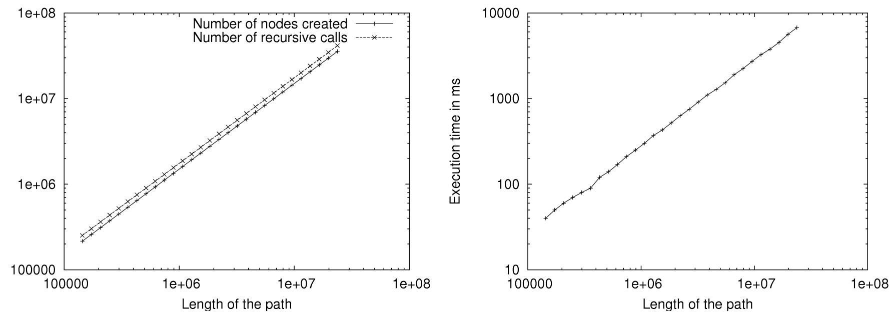
Sorting n ordered pairs can be done in $\Theta(n \log n)$ swaps or comparisons. But each swap or comparison costs $\Theta(\log n)$ clock ticks. Then the whole computation time is $\Theta(n \log^2(n))$. In our algorithm, the storage cost and computation time are both $\Theta(k)$ where k is the index of 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	$n \log^2 n$	$n \log n$
AVL 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 in both cases, and at worst $\mathcal{O}(1)$ for the first case and $\mathcal{O}(\log n)$ otherwise. The total time is hence $\mathcal{O}(n)$ in the first case, and $\mathcal{O}(n \log n)$ otherwise.

Numerical results. Our algorithms were implemented in C++ and tested on numerous examples.



The results achieved for instance with $w_n = a^n b^n$ reveal a smaller linearity constant than the constant provided in the proof of Lemma 3, and confirms their efficiency.

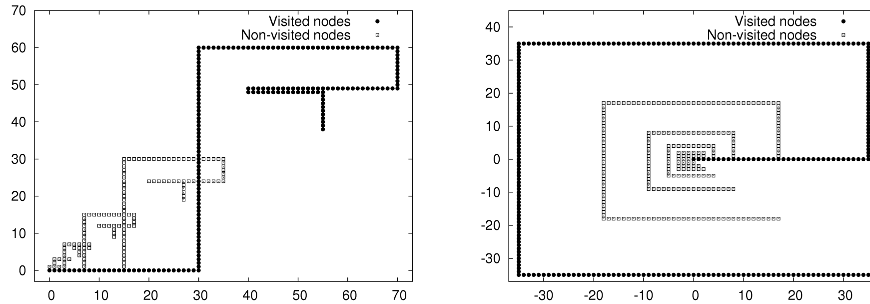


Figure 7: Planar representation of the visited nodes and non-visited nodes. Left: with input word $a^{30}b^{60}a^{40}\bar{b}^{11}\bar{a}^{30}\bar{b}a^{15}\bar{b}^{10}$. Right: with input word $a^{35}b^{35}\bar{a}^{70}\bar{b}^{70}a^{70}$.

The radix-tree built for each word corresponds, as shown in Figure 2, to points in the discrete plane $\mathbb{Z} \times \mathbb{Z}$. We illustrate in Figure 7 two examples of nodes represented in the radix-tree.

6. Multiple paths intersection

A generalization of the self-intersecting path problem is the detection of multiple paths intersection. Given a set $S = \{(w_k, p_k) \,|\, 1 \leq k \leq n\}$ where $w_k \in \{a, b, \overline{a}, \overline{b}\}^*$ and $p_i = \in \mathbb{Z}^2$ for all k, does there exist i, j, u, v with $i \neq j$ or $|u| \neq |v|$ such that u is a prefix of w_i, v is a prefix of w_j and $p_i + \vec{u} = p_j + \vec{v}$. In other words, if one draws the path coded by w_1 starting from the point p_1, the path coded by w_2 starting from p_2 and so on, do these paths intersect each other?

As mentioned at the beginning of section 5, a path may be represented starting from any point $p \in \mathbb{Z}^2$. Consider $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{B}^* \times \mathbb{B}^*$ the binary writing of the coordinates of p (using the two's complementary for negative numbers). Note that $\boldsymbol{x}$ and $\boldsymbol{y}$ are supposed to have the same length since zeros can be added to the left of a positive number's writing while in the case of a negative number, ones can be added to the left without altering its value.

Starting from the graph $\mathcal{G}_\varepsilon^{(4)}$, consider the root of the subgraph representing the quadrant where is located p. Reading $\boldsymbol{x}$ and $\boldsymbol{y}$ bit per bit from left to right, for each pair of bits $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, add a son to the previous node with label $(\boldsymbol{x}_i, \boldsymbol{y}_i)$. The last node added is ⓟ.

Finally, it suffices to adapt the algorithm `readWord` in order to start from this node ⓟ, which must be marked as *visited* at initialisation. By doing this for each pair $(w_i, p_i) \in S$ one obtains an algorithm for the multiple paths intersection problem.

Let $N = \sum_{1 \leq i \leq n} |w_i|$ and $L = \sum_{1 \leq i \leq n} \lfloor 1 + \log_2(\|p\|_\infty) \rfloor$, it is clear that this solution has a time and space complexity of $O(N + L)$. Moreover, on each node, one may choose to replace the boolean marker *visited/unvisited* by a number on $\lfloor \log(n) \rfloor$ bits and identify for which of the n paths each node is visited, 0 being the marker for non-visited nodes. Doing so, one is able to identify which pairs paths intersects each others. Of course the time and space complexity of the algorithm is increased to $O\left((N + L)\lfloor 1 + \log_2(n) \rfloor\right)$.

7. Concluding remarks

The first advantage of our algorithm is that ordering of edges can be used for avoiding labeling of both nodes and edges. Moreover, the neigbhor relation T as presented is not implemented in its symmetric form. It could be easily done since each time a neigbhor link ⓒ $\overset{\epsilon}{\dashrightarrow}$ ⓩ is added at line 7 of Algorithm 3, we can add its symmetric link ⓩ $\overset{\overline{\epsilon}}{\dashrightarrow}$ ⓒ 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.

Determining if a path w crosses itself. When a node is visited twice, deciding whether the path crosses itself or not amounts to check local conditions, describing all the possible configurations (See [8] Section 4.1).

Determining if $w \in \Sigma^n$ is the Freeman chain code of a discrete figure. It suffices

to check that the last visited node is the starting one. This does not penalize the linear algorithms for determining, for instance, if a discrete figure is digitally convex [7], or if it tiles the plane by translation [8]. In the case of a self intersecting path, it also allows the decomposition of a discrete figure in elementary components, not necessarily disjoint.

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 4, 5, 6 and 7 in Algorithm 1 by the incrementation of this counter. Then, the obsolete line 10 must be removed.

Paths in higher dimension. The graph construction extends naturally to arbitrary d-tuples in $\mathbb{B}^* \times \cdots \times \mathbb{B}^*$, for representing numbers in $\mathbb{N}^d$. Therefore, all the problems cited above can be treated in a similar way, by processing sequentially words on an alphabet $\Sigma_d = \{\epsilon_1, \bar{\epsilon}_1, \ldots \epsilon_d, \bar{\epsilon}_d\}$, of size $2d$. In the multidimensional case the trees used are no longer quadtrees but higher order trees, and in particular *octrees* for the 3-dimensional case.

Acknowledgements. The authors are grateful to Julien Cassaigne for helpful comments during the MathInfo 2010 “*Towards new interactions between mathematics and computer science*” conference in Marseille: the presentation of Section 5 was substantially simplified.

Note. A preliminary version (in French) of the results presented here appears in the doctoral thesis of Xavier Provençal [9], supported by a scholarship from FQRNT (Québec).

References

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

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

[3] 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.

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

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

[6] 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.

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

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

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

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

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

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

[13] S. Labbé, Personal communication, 2009.