Linear time and space operations on discrete paths*

Alexandre Blondin Massé Srečko Brlek
Hugo Tremblay

July 17, 2015

Abstract

We present linear time and space operations on discrete paths. First, we compute the outer hull of any discrete path. As a consequence, a linear time and space algorithm is obtained for computing the convex hull. Next, we provide a linear algorithm computing the overlay graph of two simple closed paths. From this overlay graph, one can easily compute the intersection, union and difference of two Jordan polyominoes, i.e. polyominoes whose boundary is a Jordan curve. The linear complexity is obtained by using an enriched version of a data structure introduced by Brlek, Koskas and Provençal: A quadtree for representing points in the discrete plane $\mathbb{Z} \times \mathbb{Z}$ augmented with neighborhood links, which was introduced in particular to decide in linear time if a discrete path is self-intersecting.

Keywords: Freeman code, lattice paths, radix tree, discrete sets, outer hull, convex hull, polyomino intersection, union, complement, difference.

1 Introduction

The ever-growing use of digital screens in industrial, military and civil applications gave rise to a new branch of study of discrete objects, digital

*with the support of NSERC (Canada)

geometry, where the most basic objects are pixels. In particular, their geometric properties play an essential role in the design of efficient algorithms for recognizing patterns and extracting features: these are mandatory steps for an accurate interpretation of acquired images.

Fundamental geometric operations on sets of pixels, or discrete figures have been extensively studied. For instance, algorithms computing rotations, translations, symmetries, unions, intersections, dilations or segmentations of discrete figures are well documented (see [14] for a survey of the many algorithms available). However, none of the previous method is based on encodings of discrete figures by their boundary using combinatorics on words, a field which recently led to the development of efficient tools to study digital geometry (see [1, 18]).

To illustrate the validity of this approach, consider the problem of finding the convex hull of a set of points. It is well known that for the Euclidean case, algorithms for computing the convex hull of a set $S \subset \mathbb{R}^2$ run in $\mathcal{O}(n \log n)$ time where $n = |S|$ (see [7, 15]). One can also show that such algorithms are optimal (see [8, 14, 20] for the general case). In the digital case, the situation is made surprisingly easier with the help of combinatorics on words. For instance, linear asymptotic bounds are obtained when considering discrete paths encoded by elementary steps. Indeed, Brlek et al. designed a linear time algorithm for computing the discrete convex hull of nonself-intersecting closed paths in the square grid [5]. It is based on an optimal linear time and space algorithm for factorizing a word in Lyndon words designed by Duval [11].

In this paper, we study fundamental geometric operations on connected discrete figures, or polyominoes with the help of combinatorics on words. As a first step, we describe a linear algorithm for computing the outer hull of any discrete path using the data structure described in [2] where the authors designed a linear time and space algorithm for detecting path intersection. It rests on a quadtree data structure induced by a natural radix order of $\mathbb{N} \times \mathbb{N}$. Then, each path is dynamically encoded by adding a pointer for each step of the discrete path encoded on the four letter alphabet $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$. Then, we extend those ideas to develop linear time and space algorithms for computing the overlay of two Jordan curves on $\mathbb{Z}^2$. As a byproduct, the convex hull of any discrete path, the intersection, the union and the difference of two Jordan polyominoes are computed in linear time.

2 Preliminaries

Given a finite alphabet Σ, a *word* w is a function $w : [1, 2, \ldots, n] \longrightarrow \Sigma$ denoted by its sequence of letters $w = w_1 w_2 \cdots w_n$, and $|w| = n$ is its *length*. For $a \in \Sigma$, $|w|_a$ is the number of letters a in w. The set of all words of length k is denoted by Σ^k. Consequently, $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$ is the set of all finite words on Σ where $\Sigma^0 = \{\varepsilon\}$, the set consisting of the empty word. The set Σ^* together with the operation of concatenation form a monoid called *the free monoid on* Σ.

Let w be any word. We say that the word u is a *factor* of w is there exist words x and y such that $w = xuy$. If $|x| = 0$ (resp. $|y| = 0$), then u is called a *prefix* (resp. *suffix*) of w.

There is a bijection between the set of pixels and $\mathbb{Z}^2$ obtained by mapping $(a, b) \in \mathbb{Z}^2$ to the unitary square whose bottom left vertex coordinate is (a, b). Therefore, we may consider pixels as elements of $\mathbb{Z}^2$. By definition, a *discrete set* S is a set of pixels, i.e. $S \subset \mathbb{Z}^2$. Also, two pixels are called 4-*adjacent* (resp. 8-adjacent) if their intersection is a unit segment (resp. a point). A set S is called 4-*connected* (resp. 8-connected) if for any pair of pixels $p, q \in S$, there exist pixels $p = p_0$, p_1, p_2, $\ldots$, p_{k-1}, $p_k = q$ such that p_i and p_{i+1} are 4-adjacent (resp. either 4- or 8-adjacent) for $i = 0, 1, \ldots, k-1$. Since any discrete set is a disjoint collection of 8-connected sets, we consider from now on that discrete sets are 4 or 8-connected. Also, define a *hole* of a discrete set S as a finite connected region of $\overline{S}$. Any 4-connected (resp. 8-connected) hole is called a 4-*hole* (resp. 8-*hole*).

A convenient way of representing discrete sets without hole is to use a word describing its contour (or boundary). In 1961, Freeman proposed an encoding of discrete objects by specifying their contour using the four elementary steps $(\rightarrow, \uparrow, \leftarrow, \downarrow) \simeq (\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3})$ [13]. This encoding provides an advantageous representation of discrete paths in $\mathbb{Z}^2$. A *discrete path* is a sequence of points $P = (p_1, p_2, \ldots, p_n)$ where p_i and p_{i+1} are 4-adjacent for $i = 1, 2, \ldots, n-1$. More precisely, two points p and q are called *neighbors* if $q = p + e$ for some elementary unit vector $e \in \{(1, 0), (0, 1), (-1, 0), (0, -1)\}$.

It is not hard to see that every discrete figure without hole can be represented by a closed and simple discrete path. From now on, we concentrate on the more general concept of discrete paths, referencing discrete figures without hole as "simple and closed discrete paths". For example, the discrete figure of the single pixel $(0, 0)$ is regarded as the path $((0, 0); \mathbf{0123})$.

It is worth mentioning that in the case of a closed discrete path, w is

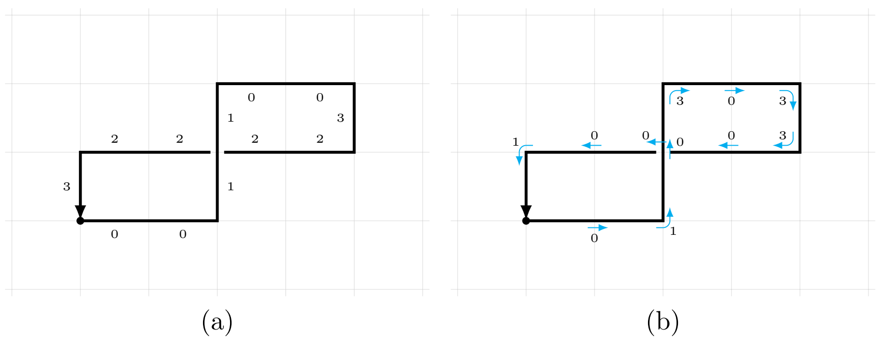


Figure 1: (a) A discrete path coded by the word $w = \mathbf{001100322223}$ and (b) its first difference word $\Delta(w) = \mathbf{01030330001}$.

unique up to a circular permutation of its letters and the sense of travel. For example, any circular permutation of the word $w = \mathbf{001100322223}$ represents the discrete path shown in Figure 1(a). One says that a word $w \in \mathcal{F}^*$ is *closed* if and only if $|w|_0 = |w|_2$ and $|w|_1 = |w|_3$. Further, w is called *simple* if it codes a nonself-intersecting discrete path, i.e. its only closed factors are ε and possibly w itself. For instance, $w = \mathbf{001100322223}$ is nonsimple and closed.

It is sometimes useful to consider encoding of paths with turns instead of elementary steps. Such encoding is obtained from the contour word $w = w_1 \cdots w_n$ by setting

$$\Delta(w) = (w_2 - w_1)(w_3 - w_2) \cdots (w_n - w_{n-1})$$

where subtraction is computed modulo 4. $\Delta(w)$ is called the *first differences word of* w. Letters of $\Delta(w) \in \mathcal{F}^*$ are interpreted via the bijection $(\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}) \simeq$ (forward, left turn, U-turn, right turn). It is worth mentioning that for any closed path w, the first difference word of w is $\Delta(w)(w_1 - w_n)$, where $\Delta(w)$ is defined as above. For example, one can verify in Figure 1(b) that $\Delta(w) = \mathbf{01030330001}$ and that it codes the turns of w.

We present two additional operations on discrete path. The first one is the usual *reversal operator*: Given $w = w_1 w_2 \cdots w_n$, we define $\widetilde{w} = w_n \cdots w_2 w_1$. Moreover, let $\widehat{w} = \widetilde{\overline{w}}$, where $\overline{\cdot}$ is the morphism defined by $0 \leftrightarrow 2$ and $1 \leftrightarrow 3$. From a geometric point of view, applying the operator $\widehat{\cdot}$ to some path w translates as traveling the path w in the opposite direction.

Clearly, every path w is contained in a smallest rectangle, or *bounding*

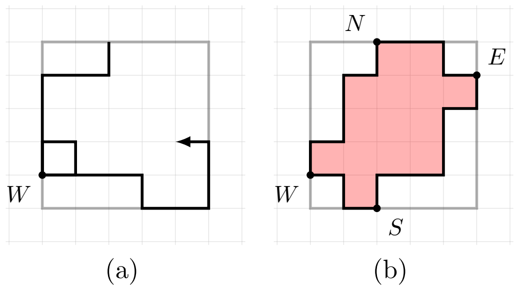


Figure 2: (a) Smallest rectangle containing a discrete path and the point W; (b) Standard decomposition of a self-avoiding closed path

box such that we can define the leftmost and then lowest point W as in Figure 2(a). W is easily obtained in linear time by keeping track of the extremum coordinates while reading the word. It is worth mentioning that in the case of a closed simple path u, this coordinate corresponds to the point W of the standard decomposition of u obtained by considering the following four extremal points of the bounding box: W (lowest on the left side), N (leftmost on the top side), E (highest on the right side) and S (rightmost on the bottom side) (see Figure 2(b)).

3 Topological graph theory

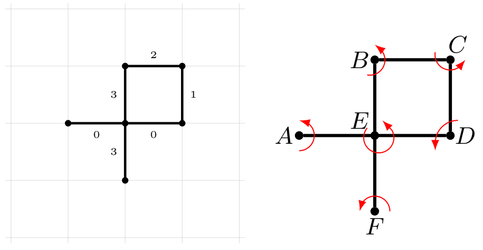
This section lays the theoretical groundwork for our study of discrete paths and objects. We begin by recalling some notions about topological graph theory (see [16] for a thorough exposition of the subject).

Let G be a graph and S a surface. Then *an embedding of* G *in the surface* S is an injective and continuous[1] function

$$I : G \longrightarrow S.$$

The components $S - G$ are called *faces* or *regions* of the embedding. Further, if I is such that every region is homeomorphic to the open disk, we say that I is a *cellular embedding*. For example, let G be any finite nonempty graph and $S = \mathbb{R}^2$. Then there does not exist cellular embedding of G in S since the infinite face is not homoemorphic to the open disk. Thus

[1]We abuse the terminology by referring to the image of the topological representation of the graph simply as G



(a) (b)

A	:	E
B	:	$C\ E$
C	:	$B\ D$
D	:	$C\ E$
E	:	$A\ F\ D\ B$
F	:	E

(c)

Figure 3: (a) The graph G_P associated with the path coded by $w = \mathbf{001233}$. (b) The counterclockwise embedding $\mathcal{G}(P)$ in $\mathbb{R}^2$ and (c) its associated rotation scheme.

from now on, we embed such graph in the sphere but draw the result in the projective plane $\mathbb{R}^2$ (with the necessary point at infinity). A graph which can be embedded in the plane is called a *planar graph*.

Now, let $G = (V, E)$ be a planar graph and $I : G \longrightarrow \mathbb{R}^2$ an embedding of G in the plane. Then I is completely determined up to homeomorphism by associating a cyclic order with the edges around each vertex of G in the following way: Begin by fixing an orientation (e.g. counterclockwise). Then for each vertex v in V, define the cyclic permutation on incident edges of v. This defines a rotation scheme on G.

Let $P = (p, w)$ be some discrete path. The *graph of* P, denoted by G_P, is the undirected graph whose vertices are the points visited by P and whose edges are connections between two consecutive points. The *graph of its image embed* in the plane $\mathbb{R}^2$ is noted $\mathcal{G}(P)$. For example, Figure 3 illustrates the path P coded by $w = \mathbf{001233}$, its graph G_P and its associated counterclockwise embedding $\mathcal{G}(P)$ in $\mathbb{R}^2$.

Finally, a closed curve in the plane $\mathbb{R}^2$ is called a *Jordan curve* if it is continuous and nonself-intersecting. An open Jordan curve is called a *Jordan arc*. Since discrete curves are composed of a finite number of straight line segments, we use the standard terminology *polygonal curve* and *polygonal arc* respectively. Thus, a *Jordan polygonal curve* is a nonintersecting closed polygonal curve and a *Jordan polygonal arc* is a nonintersecting polygonal arc. In the same spirit, we define a *Jordan polyomino* as a polyomino without hole whose boundary is a Jordan polygonal curve (see Figure 4). In other words, if P is a polyomino whose boundary is $\partial(P) = (p_1, p_2, \ldots, p_n) \in \mathbb{Z}^2 \times \mathbb{Z}^2 \times \cdots \mathbb{Z}^2$, then it is a Jordan polyomino if and only if it has no hole

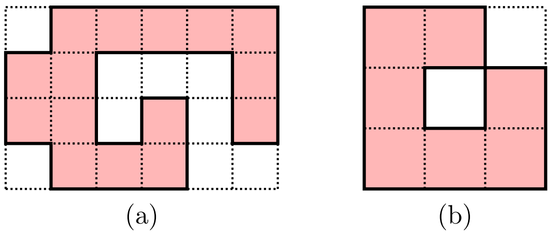


Figure 4: (a) A Jordan polyomino and (b) a non-Jordan polyomino.

and $p_i \neq p_j$ for all $1 \leq i < j \leq n$.

4 Data structure

In 2011, it was established by Brlek, Koskas and Provençal that self-intersection of paths can be verified in linear time [2]. Their idea consists in building an enriched radix quadtree so that moving by n steps in any direction on the square grid has both spatial and time complexity $\mathcal{O}(n)$. They first assume for simplicity that the whole path lies in the first quadrant $\mathbb{N} \times \mathbb{N}$, so that their data structure is restricted to points having positive coordinates. In a following discussion, they show how the data structure can be extended to $\mathbb{Z} \times \mathbb{Z}$, by constructing one quadtree per quadrant, and by describing how transitions from one quadrant to another are supported.

Here, as we are interested in computing the intersection, union and difference of two discrete regions, the first quadrant assumption is too restrictive. Therefore, we present a sligthly modified version of the quadtree, that is also enriched with edge coloring and orientation.

We use the following definitions. A *direction* is any of the four unit vectors $(1, 0)$, $(0, 1)$, $(-1, 0)$ and $(0, -1)$. Moreover, a *colored discrete path* is a triple (p, w, c), where $p \in \mathbb{Z}^2$ is called the *starting point*, $w \in \mathcal{F}^*$ is called the *direction word* and $c \in \{0, 1, \ldots, C - 1\}$ is called the *color* of the path, where C, the number of colors, is some positive integer.

When reading colored discrete paths, we are interested in storing them in an enriched radix tree. This can be done by using the following operations:

- At any time, CurrentPoint() returns the coordinates of the point where we are in the data structure. Initially, its value is the origin $(0, 0)$.

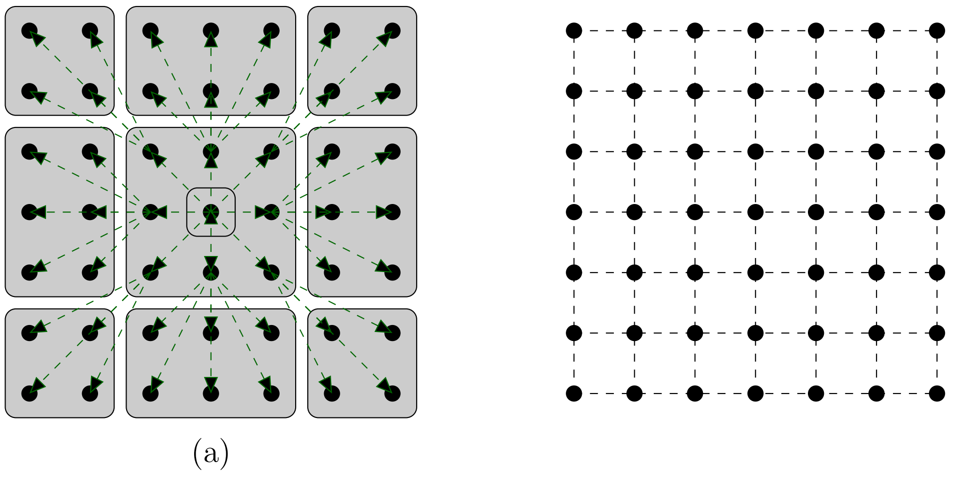


Figure 5: The two graphs on which is based the data structure. (a) The rooted tree T induced on $\mathbb{Z}^2$ by the radix relation. (b) The grid graph G on $\mathbb{Z}^2$ with the 4-adjacency relation. The data structure can be seen as a subgraph of $T \cup G$, augmented with colored arcs.

- From the current point, we can add a colored arc in any of the four directions by calling $\textsc{WriteArc}(d, c)$, where d is a direction and c is the color. At the same time, we also add a *colorless* arc in the opposite direction, which is important since every edge can be traveled in both directions. This operation also sets the current (x, y) point to $(x, y)+d$.

- As a shortcut, the operation $\textsc{WritePath}(w, c)$, where $w = w_1 w_2 \cdots w_n$ is a direction word and c is a color, is equivalent to performing the operation $\textsc{WriteArc}(w_i, c)$ for $i = 1, 2, \ldots, n$.

- At any time, it is possible to reset the current point anywhere in $\mathbb{Z}^2$, by calling $\textsc{JumpTo}(x, y)$, where $(x, y) \in \mathbb{Z}^2$.

In [2], it was proved that, in the case where all points lie in $\mathbb{N}^2$, a sequence of m calls to $\textsc{WriteArc}$ has time and space complexity $\mathcal{O}(m)$ or, equivalently, $\textsc{WriteArc}$ has amortized complexity $\mathcal{O}(1)$. For the sake of completeness, we prove here that the bound still holds when extending the space to $\mathbb{Z}^2$. As a consequence, the operation $\textsc{WritePath}(w, c)$ has complexity $\Theta(|w|)$. Moreover, we show that $\textsc{JumpTo}(x, y)$ has complexity $\mathcal{O}(|x| + |y|)$.

Algorithm 1 Finding a child of a node

Input: Any node u in the quadtree, and the indices (i, j) of the child
Output: The child node of v at indices (i, j)

```
1: function CHILD($u$ : node, $i, j$ : indices) : node
2:     if $u$.child$[i, j]$ is not defined then
3:         $u$.child$[i, j] \leftarrow$ NEWNODE$(u)$
4:     end if
5:     return $u$.child$[i, j]$
6: end function
```

The *parent* of a point $(x, y) \in \mathbb{Z}^2$ is defined by

$$\textsc{Parent}(x, y) = \left(\operatorname{sign}(x)\left\lfloor\frac{|x|}{2}\right\rfloor, \operatorname{sign}(y)\left\lfloor\frac{|y|}{2}\right\rfloor\right).$$

Conversely, if (x', y') is such that $\textsc{Parent}(x, y) = (x', y')$, then we say that (x', y') is a *child of* (x, y). It is not hard to see that

- $(0, 0)$ has eight children, namely $(\pm 1, 0)$, $(0, \pm 1)$, $(\pm 1, \pm 1)$.
- Every point $(0, y)$, where $y \in \mathbb{Z}$, has six children, which are $(0, 2y)$, $(\pm 1, 2y)$, $(0, 2y + 1)$ and $(\pm 1, 2y + 1)$ (the situation is symmetric for points of the form $(x, 0)$, where $x \in \mathbb{Z}$).
- Every point (x, y), where $x, y \neq 0$ has exactly four children, which are $(2x, 2y)$, $(2x + 1, 2y)$, $(2x, 2y + 1)$ and $(2x + 1, 2y + 1)$.

The rooted tree induced by the parent relationship described above is called *radix quadtree* (see Figure 5(a)). When moving along the four unit directions in $\mathbb{Z}^2$, we wish to create as few nodes as possible in the data structure: This can be done by inserting only the nodes that are visited by the discrete path as well as all ancestors of those nodes. Indeed, given some node u, if we want to retrieve one of its neighbors v, then it suffices to go up from u to their nearest common ancestor in the radix quadtree, and then go down from this ancestor to v. However, the problem with that strategy is that it can be expensive to navigate through the quadtree.

For this purpose, we enrich it with *neighbors links* whenever it is needed, which can be done recursively. Algorithm 1 is a simple function that returns a child of any node, while creating it if it does not already exist in

Algorithm 2 Finding a neighbor of a node

```
Input: Any node $u$ in the quadtree, the coordinates $(x, y)$ of this node and
    the direction $d$ pointing toward the neighbor
Output: The neighbor node of $u$ in direction $d$
 1: function NEIGHBOR($u$ : node, $(x, y)$ : point, $d$ : direction) : node
 2:     if $u$.neighbor[$d$] is not defined then
 3:         $(x', y') \leftarrow (x, y) + d$
 4:         if $(x, y) = (0, 0)$ then
 5:             $v \leftarrow$ CHILD($u, d$)
 6:         else if $(x', y') = (0, 0)$ then
 7:             $v \leftarrow u$.parent
 8:         else
 9:             Let $i$ be $-1$ if $x' = -1$, and $x' \bmod 2$ otherwise
10:             Let $j$ be $-1$ if $y' = -1$, and $y' \bmod 2$ otherwise
11:             if $(x, y)$ and $(x', y')$ share the same parent then
12:                 $v \leftarrow$ CHILD($u$.parent, $i, j$)
13:             else
14:                 $v \leftarrow$ CHILD(NEIGHBOR($u$.parent, PARENT($x, y$), $d$), $i, j$)
15:             end if
16:         end if
17:         $u$.neighbor[$d$] $\leftarrow v$
18:         $v$.neighbor[$-d$] $\leftarrow u$
19:     end if
20:     return $u$.neighbor[$d$]
21: end function
```

the data structure. The children are indexed by a couple (i, j) whose semantics is described in Figure 6. We also assume that there exists a function NEWNODE(u) which creates a new node and adds u as its parent.

Algorithm 2 describes the function NEIGHBOR($u, (x, y), d$), which returns the neighbor v of u in direction d. The point represented by u is (x, y) (notice that it is not mandatory to store the values (x, y) in the data structure, as it may be deduced from the position of u in it). There are five possible scenarios when accessing the neighbor of a node :

1. The neighbor link already exists, so that we can use it directly.

2. The current node is the root (lines 4-5). Then the neighbor is merely

its child in direction d, and we can use the direct link.

3. The neighbor is the root (lines 6-7).
4. The node and its neighbor share the same parent in the radix tree (lines 11-12). Then we can directly access the neighbor and link it.
5. The node and its neighbor do not share the same parent (lines 13-14). In this case, we recursively try to link the parents of both nodes. When this is done, we retrieve the neighbor, which is its parent's neighbors's child. It remains to link it directly.

It follows from this study that the operation NEIGHBOR takes constant time except in the 5th case where several recursive calls are made. However, we show that this case is rare:

Theorem 4.1. *Starting from the origin, the operation* WRITEPATH(w, c) *has complexity* $\Theta(|w|)$ *or, equivalently, the operation* WRITEARC *has amortized cost* $\mathcal{O}(1)$.

Proof. The proof is essentially the same as in [2]. We include it for the sake of completeness and to confirm that it extends for a quadtree on $\mathbb{Z}^2$.

First, notice that the complexity of WRITEPATH(w, c) is proportional to the number of nodes in the quadtree, as all operations linking the nodes are done in constant time. Therefore, if m is the number of nodes, then it only remains to show that $m = \mathcal{O}(n)$.

For any discrete path p, let POINTS(p) be the set of points in $\mathbb{Z}^2$ visited by p. Moreover, if $i \geq 0$, let PARENT$^i(p)$ be the discrete path whose sequence of points is described by the parents of the points of p. Then the set X of points of $\mathbb{Z}^2$ appearing in the data structure is

$$X = \bigcup_{i=0}^{h} \textsc{Points}(\textsc{Parent}^i(p)) \tag{1}$$

where PARENT^i denotes the function PARENT applied i times and h is the height of the quadtree.

But any sequence of five consecutive points in a discrete path can have at most four parents (this is Lemma 2 of [2]), i.e. at least two points share the same parent. In other words, for any discrete path q,

$$|\textsc{Points}(\textsc{Parent}(q))| \leq \frac{4}{5}|\textsc{Points}(q)|,$$

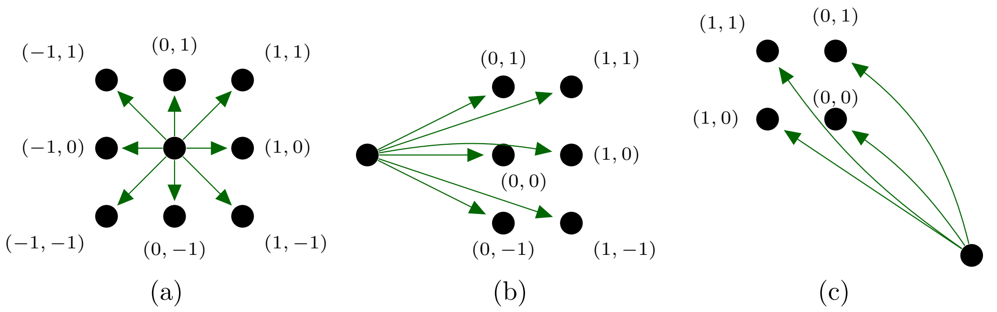


Figure 6: The valid indices for accessing children of nodes. (a) From the root, (b) from an axis point and (c) from any other point.

so that

$$|\textsc{Points}(\textsc{Parent}^i(p))| \leq \left(\frac{4}{5}\right)^i |\textsc{Points}(p)|.$$

This implies that

$$m = |X| = \sum_{i=0}^{h} |\textsc{Points}(\textsc{Parent}^i(p))| \leq \sum_{i=0}^{\infty} n \left(\frac{4}{5}\right)^i = 5n = \mathcal{O}(n),$$

as desired. □

The complexity of JumpTo is immediate.

Corollary 4.2. *The operation* JumpTo(x, y) *has complexity* $\mathcal{O}(|x| + |y|)$ *in the worst case.*

Proof. In order to reset the current point to (x, y), it suffices to move $|x|$ times in the direction of $(\mathrm{sign}(x), 0)$ followed by $|y|$ times in the direction $(0, \mathrm{sign}(y))$, while updating the parent-child links as well as the neighbors links (but without coloring the arcs). □

5 Outer and convex hull

We recall from topology that given a set S, the boundary ∂S is the set of points in the closure of S, not belonging to the interior of S. Now, let S be a 8-connected discrete set. The *outer hull of* S, denoted $\mathrm{Hull}(S)$ is the

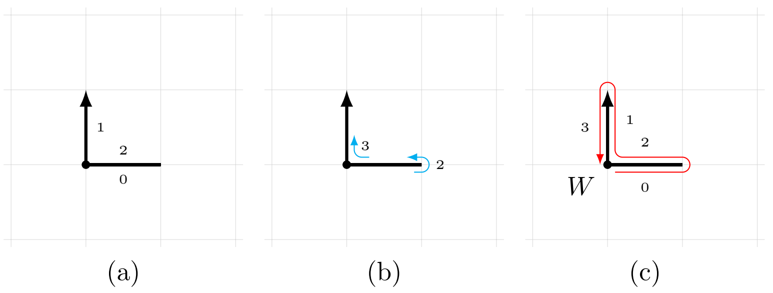


Figure 7: (a) The path $w = \mathbf{021}$, (b) its first difference word $\Delta(w) = \mathbf{23}$ and (c) its outer hull $\text{Hull}(w) = \mathbf{0213}$.

boundary of the intersection of all discrete sets without hole containing S, i.e. the nonself-intersecting path following the exterior contour of S. Definition 5.1 extends the notion of outer hull to any discrete path.

Definition 5.1. *Let P be any discrete path. Then, the outer hull of P, denoted by* $\text{Hull}(P)$ *is the outer face of the embedded graph $\mathcal{G}(P)$.*

The difference between Definition 5.1 and the preceding one lies in the use of the embedding of P in the plane instead of a discrete set to describe the outer hull. This choice is not arbitrary as it allows the computation of the outer hull even when part of the discrete path are degenerate line segments (i.e. Euclidean sets of null measure). For example, Figure 7 illustrates the outer hull of the path coded by $w = \mathbf{021}$. Remark that using Definition 5.1, the boundary of discrete line segments are coded by closed words, e.g. the outer hull of the horizontal line segment coded by 0 is coded by 02.

This ensures that Definition 5.1 is a convenient generalization of the outer hull to discrete paths. Indeed, if P codes the boundary of a discrete set S, then P is simple and closed by definition. This gives $P = \text{Hull}(P)$ and since $\text{Hull}(S)$ is the boundary $\partial(S)$ of S by definition, we have

$$\text{Hull}(S) = \partial(S) = P = \text{Hull}(P).$$

Since there is a bijection between discrete paths in $\mathbb{Z}^2$ and words on $\mathcal{F}$ coupled with a starting point, we identify P with its coding word w and we write $\text{Hull}(w)$ instead of $\text{Hull}(P)$.

We now recall some basic notions concerning digital convexity, for which a detailed exposure appears in [5, 18]. Let S be an 8-connected discrete set. S is *digitally convex* if it is the Gauss digitization of a convex subset R of $\mathbb{R}^2$, i.e. $S = \text{Conv}(R) \cap \mathbb{Z}^2$. The *convex hull of S*, denoted $\text{Conv}(S)$ is the intersection of all convex sets containing S. In the case of a closed simple path w, $\text{Conv}(w)$ is given by the Spitzer factorization of w (see [5, 19]). Given $w = w_1w_2\cdots w_n \in \{0,1\}^*$, one can compute the NW part of this factorization as follows: Start with the list $(b_1, b_2, \ldots, b_n) = (w_1, w_2, \ldots, w_n)$. If the slope $\rho(b_i) = |b_i|_1/|b_i|_0$ of b_i is strictly smaller than that of b_{i+1} for some i, then

$$(b_1, b_2, \ldots, b_k) = (b_1, \ldots, b_{i-1}, b_ib_{i+1}, b_{i+2}, \ldots, b_k).$$

By repeating this process until it is no longer possible to concatenate any words, one obtains the Spitzer factorization of w. The NE, SE and SW parts of the factorization are obtained by permuting the alphabet.

5.1 Outer hull algorithm

Let $w \in \mathcal{F}^*$ be a discrete path and G_w its graph representation. Remark that the application $g : w \mapsto G_w$ is not bijective since it is not injective (for example, $u = \mathbf{0}$ and $v = \mathbf{02}$ admits the same graph). Now, recall from Section 2 that the embedding $\mathcal{G}(w)$ of G_w in $\mathbb{R}^2$ gives rise to a rotation scheme (provided we fix an orientation). We use this embedding to compute the outer hull of w (i.e. the outer face of $\mathcal{G}(w)$): Fix an orientation $\mathcal{O}$ of the surface $\mathbb{R}^2$ and let $e_i = (u, v)$ be an arc from vertex u to v in $\mathcal{G}(w)$ such that e_i is an edge of the outer face of the embedding $\mathcal{G}(w)$ and such that e follows the fixed orientation $\mathcal{O}$. Next, compute $e_{i+1} = (v, \sigma_v(u))$ where σ_v is the cyclic permutation associated with v in $\mathcal{G}(w)$. By letting $\mathcal{O}$ be the counterclockwise orientation, one can iterate this process to obtain the outer face of $\mathcal{G}(w)$. For example, using the rotation scheme defined for $w = \mathbf{001233}$ in Figure 3(c) and starting with the arc (A, E), one computes the sequence of arcs

$$(A, E),\ (E, F),\ (F, E),\ (E, D),\ (D, C),\ (C, B),\ (B, E),\ (E, A)$$

which corresponds to the outer hull of w (see Figure 8).

The correctness of this method follows from the so-called "right-hand rule" or "wall follower algorithm" for traversing mazes. Indeed, given an arc (u, v), taking the adjacent arc $(v, \sigma_v(u))$ amounts to "turning right" at vertex v (see Figure 8(c)). The underlying principle of our algorithm is thus to walk along the path, starting at an origin point on the outer hull and turning systematically right at each intersection and returning to the origin

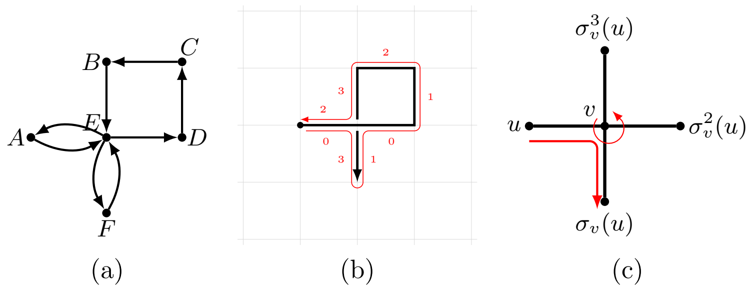


Figure 8: (a) The sequence of arcs obtained by using the rotation scheme of Figure 3(c); (b) The outer hull of $w = \mathbf{001233}$; (c) The sequence (u, v), $(v, \sigma_v(u))$ corresponds to a right turn in the graph of a path, provided the orientation is counterclockwise

point. The preceding discussion guarantees that the resulting walk is then precisely the outer hull of w.

To efficiently implement this procedure, several problems must be addressed. First, as stated before, the walk needs to start on a coordinate of the outer hull, otherwise the resulting path may not describe the correct object. This can be solved by choosing the point W associated with the contour word w as the starting point.

Secondly, whenever a path returns to W before continuing on (the simplest of which is the path coded by $w = \mathbf{021}$, see Figure 7), one must ensure that the algorithm does not stop until every such sub-path has been explored. An easy solution for managing that situation is to keep a list of all neighbors of W that are in the path P associated with w. This list has at most two elements since no vertex in P is located below or left of W.

Finally, one needs to recognize intersections and decide of the rightmost turn. We solve this problem by using the quadtree structure described in Section 4. This leads to Algorithm 3 for computing the outer hull of a discrete path w, which proceeds as follows.

First, the quadtree G associated with w is built starting from W. Then, the graph G is traversed from its root W, following the path represented by w. At every intersection c, we need to:

(a) extract the letter α associated with the vector $\overrightarrow{cv}$ for each neighbor v of c;

(b) determine the turn associated with each v, that is $\Delta(w_c \cdot \alpha)$;

Algorithm 3 Computing the outer hull of a discrete path

```
Input: A word $w \in \mathcal{F}^*$ coding a discrete path
Output: A simple word $w' \in \mathcal{F}^*$ describing $\mathrm{Hull}(w)$
 1: Let $W$ be the leftmost lowest coordinate on the bounding box of $w$
 2: Construct the quadtree $G$ associated with $w$ rooted in $W$
 3: Let $N$ be the set of all visited neighbors of $W$
 4: $c \leftarrow W + (1,0)$ if it is in $N$ or $W + (0,1)$ otherwise
 5: $w' = \textsc{Freeman}(c - W)$
 6: while $c \neq W$ or $N \neq \varnothing$ do
 7:     $t = 2 \bmod 4$
 8:     for each neighbor v of c do
 9:         if $[\textsc{Freeman}(v-c) - \textsc{Lst}(w') + 1] \bmod 4 \leq [t+1] \bmod 4$ then
10:             $t \leftarrow \textsc{Freeman}(v-c) - \textsc{Lst}(w')$
11:             $next \leftarrow v$
12:         end if
13:     end for
14:     $w' = w' \cdot \textsc{Freeman}(next - c)$
15:     $N.\textsc{Remove}(c)$
16:     $c \leftarrow next$
17: end while
18: return $w'$
```

(c) choose the rightmost one, that is the closest to 3.

This procedure ends when returning to the point W.

Theorem 5.2 (Correctness of Algorithm 3). *For any word $w \in \mathcal{F}^*$, Algorithm 3 returns* $\mathrm{Hull}(w)$.

Proof. Let $k = |\,\mathrm{Hull}(w)|$. We use the following loop invariant:

> At the start of the i^{th} iteration of the while loop in Line 6, w' is a prefix of length i of the contour word associated with $\mathrm{Hull}(w)$.

The invariant holds the first time Line 6 is executed, since at that time, w' is the first step of the outer hull of w computed at Line 5. Now, assume the invariant holds before the i^{th} iteration of the loop. Then, Lines 8 to 13 find the rightmost turn at the current coordinate c. Then in Line 14, w' is concatenated with the step of this turn. By the right-hand rule for solving simply connected maze, considering rightmost turns yields coordinates on the outer hull of w. Consequently, at the end of the iteration, w' is a prefix of the contour word associated with $\mathrm{Hull}(w)$ of length $i+1$. Finally, at the end of the loop, w' is a prefix of the contour word associated with $\mathrm{Hull}(w)$ of length k, that is $w' = \mathrm{Hull}(w)$. Note that since any neighbor of W is on $\mathrm{Hull}(w)$, Line 15 clearly removes every element from N yielding, at termination, an empty set. $\square$

The complexity of Algorithm 3 is linear both in time and space.

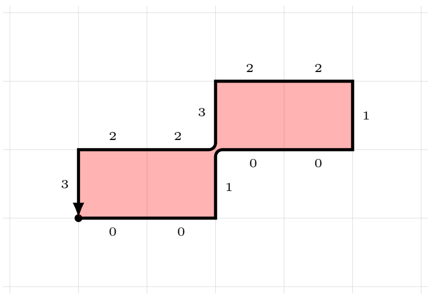
Theorem 5.3. *Let $w \in \mathcal{F}^*$. Then Algorithm 3 computes $\mathrm{Hull}(w)$ in $\Theta(|w|)$ space and time.*

Proof. First, the quadtree structure is constructed in linear time with respect to $|w|$ (see [2]). Also, as stated before, the point W is easily computed in linear time. Consequently, computations in Line 2 are performed in linear time. Next, Line 1, 4 and 5 each take constant time. Moreover, the set N is constructed in linear time by accessing neighborhood informations of the root in the quadtree structure, so Line 3 takes linear time. Now, since any coordinate has at most four neighbors, the **for** loop in Line 8 is executed at most four times per iteration of the **while** loop. Line 15 takes constant time. This is due to the fact that N contains at most two elements. Since instructions in Line 7, 9, 10, 11, 14 and 16 all are computed in constant time, at most $\mathcal{O}(k)$ computations occur during the execution of the **while** loop, thus making Algorithm 3 linear in time. Finally, the quadtree structure needs linear space in k, since it only uses an enriched quadtree as additional space. $\square$

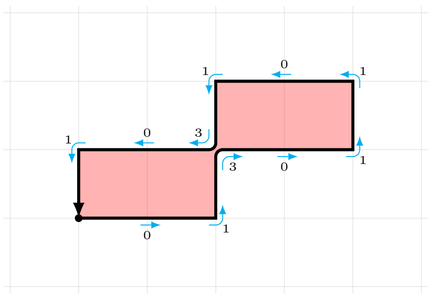
Example 5.4. *Consider the word $w =$ **001100322223**. Then, Algorithm 3 yields $w' =$ **001001223223** (see Figure 9). One can easily verify that w' is a simple path describing the outer hull of w, so* $\mathrm{Hull}(w) = w'$.

Our algorithm was implemented using the C++ programming language and tested with numerous examples (see Figure 10). The source code is available on demand.

Finally, we show how Algorithm 3 can be used to compute in linear time and space the convex hull of any discrete path. It relies on the following rather obvious result:



(a) $w' = 001001223223$



(b) $\Delta(w') = 01301101301$

Figure 9: Outer hull of $w = 001100322223$

Proposition 5.5. *Let $w \in \mathcal{F}^*$ be a boundary word coding a discrete path. Then,*

$$\mathrm{Conv}(w) = \mathrm{Conv}(\mathrm{Hull}(w)).$$

Proof. If w is simple, then $\mathrm{Hull}(w) = w$ so the claim holds. Now, suppose w is nonsimple. Then by definition, $\mathrm{Hull}(w)$ is the boundary of w. Since, $\mathrm{Conv}(w)$ is the intersection of all convex sets containing w, it must also contain $\mathrm{Hull}(w)$ and thus $\mathrm{Conv}(w) = \mathrm{Conv}(\mathrm{Hull}(w))$. $\square$

Recall that $\mathrm{Hull}(w)$ is nonself-intersecting for any path w. Proposition 5.5 then yields a very simple procedure for computing the convex hull of a discrete path using Brlek et al. simple path convex hull algorithm (see [5]):

1. Start by computing $\mathrm{Hull}(w) = w'$;
2. compute $\mathrm{Conv}(w')$.

It is clear that the preceding procedure computes the convex hull of a discrete path in linear time and space. Indeed, we showed in Section 5.1 that the first step is computed in linear time and space. Furthermore, it is shown in [5] that the second step is computed in a similar fashion.

6 Overlay graph of two polyominoes

Let $P_1 = (p_1; w_1)$ and $P_2 = (p_2; w_2)$ be two Jordan polyominoes (i.e. P_1 and P_2 are polyominoes without hole such that their boundary are Jordan curves). Recall from Section 3 that given a planar graph G embedded in the plane $\mathbb{R}^2$, the faces of G are the components $\mathbb{R}^2 - G$. Consider the graphs G_{P_1}

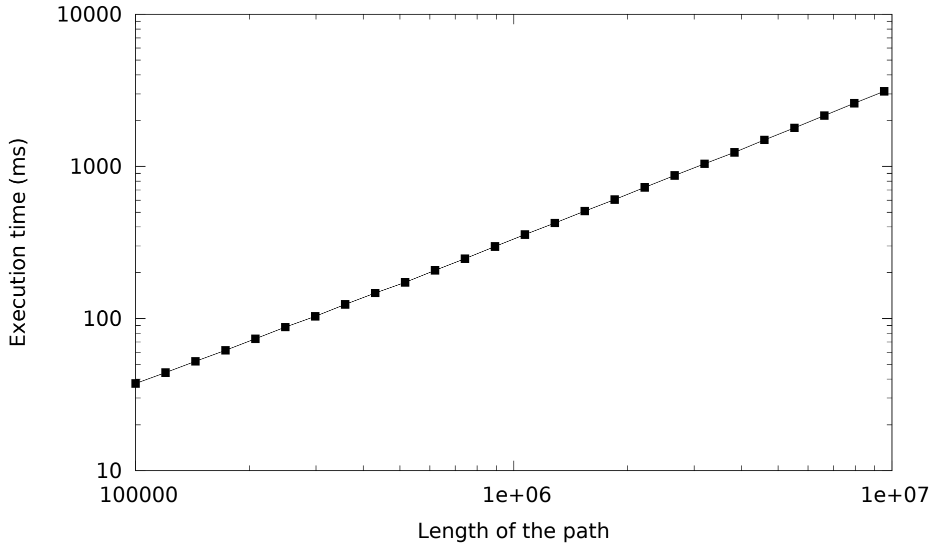


Figure 10: Running time of Algorithm 3 for random discrete paths of length 10^5 to 10^7, with each point representing the mean running time of 100 random discrete paths of same length

and G_{P_2}. It is clear that each graph admits exactly two faces: Its interior and its exterior. Further, these two faces are completely determined by a walk on their boundary. We now wish to compute the faces of the so-called *overlay graph* $O(P_1, P_2)$, that is the graph such that there is a face F in $O(P_1, P_2)$ if and only if there are faces F_1 in G_{P_1} and F_2 in G_{P_2} such that F is a maximal connected subset of $F_1 \cap F_2$ [2]. For example, Figure 11 illustrates two Jordan polyominoes represented by their boundary and their associated overlay graph.

The following result is a corollary of the well-known *Jordan curve theorem for polygonal arcs*: Any Jordan curve divides the plane into two distinct region; The interior and the exterior (see [12] for one of the numerous proof).

Proposition 6.1. *Let γ be a Jordan polygonal curve in $\mathbb{R}^2$ and $p, q \in \gamma$. Let ξ be a Jordan polygonal arc from p to q such that ξ is in the interior of γ. Then, $\xi \cup \gamma$ divides $\mathbb{R}^2$ into exactly three regions whose boundary are Jordan polygonal curves.*

[2]This definition comes from [10] where the authors give an $O(n \log n)$ algorithm for computing the overlay of two regions in the plane.

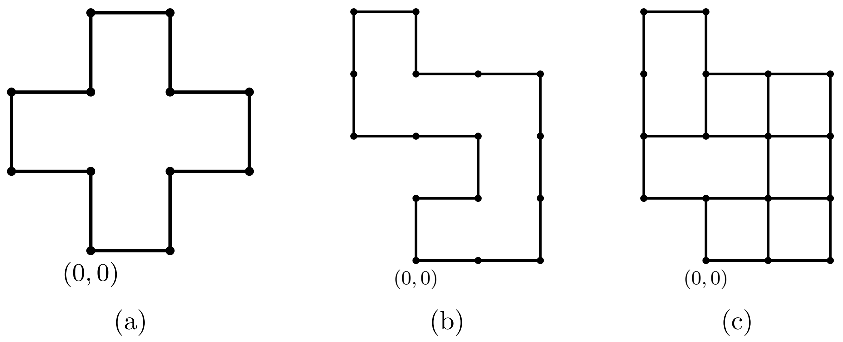


Figure 11: (a) The graph of $P_1 = ((0,0); \mathbf{010121232303})$. (b) The graph of $P_2 = ((0,0); \mathbf{0011122123300323})$. (c) The overlay graph $O(P_1, P_2)$

Now, applying Proposition 6.1 to the overlay $O(P_1, P_2)$ of two intersecting Jordan polyominoes, it follows that the boundary of all faces of $O(P_1, P_2)$ is a Jordan curve (in fact, it is a Jordan polygonal curve).

6.1 Overlay algorithm

Given two Jordan polyominoes $P_1 = (p_1; w_1)$ and $P_2 = (p_2; w_2)$, we now give a linear time and space algorithm for computing the overlay of P_1 and P_2. Like Algorithm 3, it is based on the wall-follower algorithm for traversing mazes and makes extensive use of the enriched radix tree data structure described in Section 4. We also make the assumption that w_1 and w_2 codes their respective boundary in a counterclockwise fashion. This is done without loss of generality since one can compute the orientation of a word in linear time using its turning number [4, 9, 17].

Remark that the overlay $O(P_1, P_2)$ does not need to be connected. This occurs when the boundaries of P_1 and P_2 do not intersect, so that these cases need to be handled separately.

We can now describe Algorithm 4. As a first step, we check if the bounding boxes are disjoint. If it is the case, then the polyominoes are also disjoint. Otherwise, we compute the enriched radix tree T associated with P_1 and P_2 described in Section 4. Next, we check if the boundary of P_1 and P_2 intersects. In the case of nonintersecting boundaries, the overlay $O(P_1, P_2)$ is not connected and there are three cases to consider:

(i) The bounding box of P_1 is included in the bounding box of P_2, which

Algorithm 4 Computing the overlay of two Jordan polyominoes.

```
Input: Two positively oriented Jordan polyominoes $P_1$ and $P_2$
Output: The set of faces of the overlay graph $O(P_1, P_2)$.
 1: function OVERLAY($P_1 = (p_1; w_1)$ and $P_2 = (p_2; w_2)$)
 2:     if BOUNDINGBOX($P_1$) $\cap$ BOUNDINGBOX($P_2$) $= \emptyset$ then
 3:         return $\{(P_1, \{0\}), (P_2, \{1\}), (\widehat{P_1}, \widehat{P_2}, \emptyset)\}$
 4:     else
 5:                                   $\triangleright$ We insert $P_1$ and $P_2$ in the data structure
 6:         $T \leftarrow$ ENRICHEDRADIXTREE()
 7:         $T$.WRITEPATH($w_1, 0$)
 8:         $T$.JUMPTO($p_2 - p_1$)
 9:         $T$.WRITEPATH($w_2, 1$)
10:         if no point of $T$ is visited at least twice then
11:             if BOUNDINGBOX($P_1$) $\subset$ BOUNDINGBOX($P_2$) then
12:                 return $\{(P_1, \{0,1\}), (P_2, \widehat{P_1}, \{1\}), (\widehat{P_2}, \emptyset)\}$
13:             else if BOUNDINGBOX($P_2$) $\subset$ BOUNDINGBOX($P_1$) then
14:                 return $\{(P_2, \{0,1\}), (P_1, \widehat{P_2}, \{0\}), (\widehat{P_1}, \emptyset)\}$
15:             else
16:                 return $\{(P_1, \{0\}), (P_2, \{1\}), (\widehat{P_1}, \widehat{P_2}, \emptyset)\}$
17:             end if
18:         else
19:             return NONDISJOINTOVERLAY($T$)
20:         end if
21:     end if
22: end function
```

means that $P_1 \subset P_2$;

(ii) the bounding box of P_2 is included in the bounding box of P_1, which means that $P_2 \subset P_1$;

(iii) the polyominoes are disjoint.

Otherwise, in the case of intersecting boundaries, the connected overlay $O(P_1, P_2)$ is computed using the function NONDISJOINTOVERLAY described in Algorithm 5.

The function NONDISJOINTOVERLAY is implemented as follows. First, we choose (p, p') an arc of T and then we follow the counterclockwise path induced by this arc, taking the leftmost turn at each intersection until returning to the point p. While following the path, we delete any visited arc from T as the path is followed. Since each neighborhood arc of T belongs to the boundary of exactly one face of the overlay graph, we obtain a list of Jordan polygonal curves, describing the boundaries of all the faces of $O(P_1, P_2)$. It remains to identify each face to its associated poylomino(es). This is easily achieved by keeping track of the colors of the arcs in any given path. This procedure is described more formally in Algorithm 5. It uses a "first-in-first-out" queue to keep track of the various visited points[3].

Theorem 6.2 (Correctness of Algorithm 5)**.** *For any two Jordan polyominoes P_1 and P_2 intersecting on their boundary, Algorithm* 5 *returns all faces of $O(P_1, P_2)$.*

Proof. The first time Line 4 is executed, no face of the overlay graph has been computed yet. Lines 7 and 8 initialize a new face by choosing an outgoing arc at the current point p. Now, two cases can occur depending on whether p has any outgoing arc.

If p has at least one outgoing arc, then we follow the path induced by this outgoing arc, turning left at intersections, while keeping track of the edge colors (Lines 10 to 14). By the wall-follower algorithm, this defines the boundary of a face of $O(P_1, P_2)$. It is worth noticing that this face has not been previously computed since each arc appears in the boundary of exactly one face of $O(P_1, P_2)$ and arcs are deleted as they are followed (Line 13).

Otherwise, p is simply dequeued from Q and no new face is computed. Finally, at the end of the loop, all outgoing arcs for all points in the structure have been considered exactly once. Since each arc in the structure belongs to the boundary of exactly one face (edges are represented by arcs in both directions) of the overlay graph, all such faces have been computed. □

Corollary 6.3 (Correctness of Algorithm 4)**.** *For any two Jordan polyominoes P_1 and P_2, Algorithm* 4 *returns all faces of $O(P_1, P_2)$.*

Proof. Two cases can occur depending on whether $O(P_1, P_2)$ is connected or not. It is immediate that $O(P_1, P_2)$ is connected if and only if P_1 and P_2 intersect on their boundary. In the opposite case, it remains to verify the two polyominoes relative positions. Since this is easily done by keeping track of extremum coordinates, we have the result. □

[3]A similar algorithm is presented in [10]. It utilizes a doubly-linked list and a plane sweep method to achieve a $\mathcal{O}(n \log n)$ time bound for the Euclidean case.

Algorithm 5 Computing the overlay of intersecting Jordan polyominoes

Input: An enriched radix tree containing two Jordan polyominoes
Output: The set of faces of the overlay graph $O(P_1, P_2)$.

```
1: function NonDisjointOverlay($T$ : enriched radix tree)
2:     $O \leftarrow \emptyset$
3:     Let $Q$ be a queue containing all points of $T$;
4:     while $Q$ is not empty do
5:         $s \leftarrow$ the first element of $Q$;
6:         if $s$ has at least one outgoing arc $(s, s')$ then
7:             $F \leftarrow$ NewFace$(s, \varepsilon, \emptyset)$
8:             $e \leftarrow (p, p') \leftarrow (s, s')$
9:             repeat
10:                $F$.Colors.Add$(e.color())$
11:                $F$.Word $\leftarrow$ $F$.Word $\cdot$ Freeman$(p' - p)$
12:                $p'' \leftarrow$ Leftmost$(e)$
13:                $T$.Delete$(e)$
14:                $e \leftarrow (p, p') \leftarrow (p', p'')$
15:            until $p = s$
16:            $O$.Add$(F)$
17:        else
18:            $Q$.Dequeue$(s)$;
19:        end if
20:    end while
21:    return $O$
22: end function
```

Example 6.4. *Let*

$$\begin{aligned} P_1 &= ((0,0); \mathbf{0^3 1^5 2 3^4 2 1 1 2 3^3}) \\ P_2 &= ((0,2); \mathbf{0 1 0^3 1 2^3 1 2 3^3}). \end{aligned}$$

The overlay $O(P_1, P_2)$ is represented in Figure 12. Since their boundary intersects, Algorithm 5 applies. After computations, we obtain the following

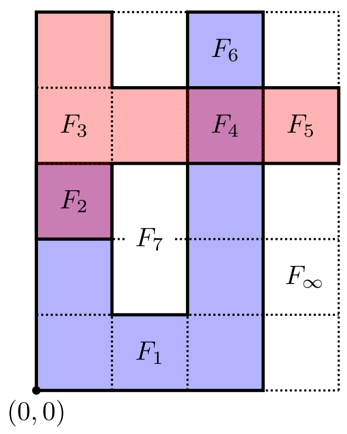


Figure 12: The faces of the overlay of $P_1 = ((0,0); \mathbf{0}^3\mathbf{1}^5\mathbf{2}\mathbf{3}^4\mathbf{21123}^3)$ and $P_2 = ((0,2); \mathbf{010}^3\mathbf{12}^3\mathbf{123}^3)$ after using Algorithm 5.

faces for the overlay:

$$\begin{aligned}
\textsc{Face}_\infty &= \{(0,0), 1^5 0301030323^3 2^3, \emptyset\}\\
\textsc{Face}_1 &= \{(0,0), 0^3 1^3 23321233, \{\textsc{blue}\}\}\\
\textsc{Face}_2 &= \{(0,2), 0123, \{\textsc{blue}, \textsc{red}\}\}\\
\textsc{Face}_3 &= \{(0,3), 00121233, \{\textsc{red}\}\}\\
\textsc{Face}_4 &= \{(2,3), 0123, \{\textsc{blue}, \textsc{red}\}\}\\
\textsc{Face}_5 &= \{(3,3), 0123, \{\textsc{red}\}\}\\
\textsc{Face}_6 &= \{(2,4), 0123, \{\textsc{blue}\}\}\\
\textsc{Face}_7 &= \{(1,1), 011233, \emptyset\}.
\end{aligned}$$

We now show that Algorithm 5 has linear time and space complexity.

Theorem 6.5. *Let $P_1 = (p_1, w_1)$ and $P_2 = (p_2, w_2)$ be two closed and simple discrete paths. Then the overlay graph of G_{P_1} and G_{P_2} can be computed in $\mathcal{O}(|w_1| + |w_2|)$ time and space.*

Proof. All conditions in Lines 2, 10, 11 and 13 are verified in linear time since the data structure is also computed in linear time (Line 6). Moreover, Line 8 has linear cost by Corollary 4.2. All other lines are done in constant time.

It only remains to count the number of times the **while** and **repeat** loops are repeated in Algorithm . First, notice that no Enqueue operations are performed. Moreover, elements of Q are dequeued whenever they have no outgoing arc, but these arcs are removed at Line 13. Therefore, the **while** loop is repeated $\mathcal{O}(|w_1| + |w_2|)$ times.

Also, the body of the **repeat** loop is executed once per arc in the data structure. But the number of arcs is at most $2(|w_1| + |w_2|)$, which implies that the overall time for the **repeat** loop is $\mathcal{O}(|w_1| + |w_2|)$ times.

Finally, as Algorithm 4 relies almost entirely on the enriched radix quadtree structure for storing information, it takes linear space. □

We end this section by establishing a simple relation between the number of faces of the overlay graph and the number of intersection points.

Let e, e_1 and e_2 be the number of edges of $O(P_1, P_2)$, G_{P_1} and G_{P_2} respectively. Since P_1 and P_2 are Jordan polyominoes, we have

$$v_1 = e_1 = |w_1| \text{ and } v_2 = e_2 = |w_2|.$$

Let i be the number of intersection points of the overlay, that is points with more than two neighbors. Then,

$$v = v_1 + v_2 - i \text{ and } e \leq e_1 + e_2.$$

From Euler's formula for planar graphs, we know that the number f of faces of a graph is given by the equation

$$f = 2 - v + e.$$

Therefore, the number of faces of $O(P_1, P_2)$ is

$$\begin{aligned} f &= 2 - v + e \\ &\leq 2 - (v_1 + v_2 - i) + e_1 + e_2 \\ &= 2 + i. \end{aligned}$$

6.2 Other operations

If polyominoes are represented by boolean matrices, it is very easy to compute their intersection, reunion and difference: It suffices to apply the corresponding logical operation bitwise. Here, we are interested in computing them with respect to the boundary of the polyominoes. Similar algorithms have been proposed to compute the *area* of the intersection, union and difference of two polyominoes based on a discrete version of Green's algorithm [3], but linear algorithms to explicitly describe the resulting polyominoes have never been provided in the literature.

From Algorithm 4, we can easily design such algorithms, at least in the case where the two polyominoes P_1 and P_2 are Jordan polyominoes. However, the result may not be a Jordan polyomino. In general, such operations yield a family of polyominoes (with possibly some holes, see Figure 13 for example).

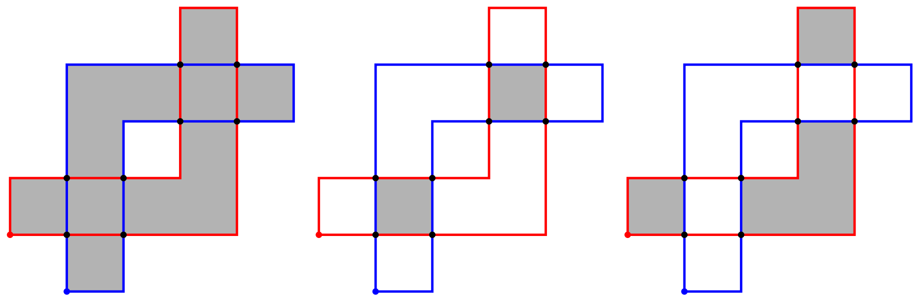

(a) $P_1 \cup P_2$ (b) $P_1 \cap P_2$ (c) $P_1 - P_2$

Figure 13: Various boolean operations on two Jordan polyominoes P_1 (in red) and P_2 (in blue)

To obtain those boolean operations, we start by computing $O(P_1, P_2)$. Then, we choose regions of the overlay corresponding to the desired operation: $P_1 \cap P_2$ is the set of faces whose color set is $\{0, 1\}$ (i.e. belonging to P_1 and P_2), $P_1 \cup P_2$ is the set of faces whose color set is nonempty (i.e. belonging to P_1 or P_2) and $P_1 - P_2$ is the set of faces whose color is $\{0\}$ (i.e. belonging to P_1 but not to P_2). It follows from Section 6.1 that the intersection, union and difference of Jordan polyominoes is computed in linear time and space.

7 Concluding remarks

In the previous sections, we showed that linear time and space algorithms can be obtained by relying on one powerful data structure introduced by Brlek, Koskas and Provençal in [2]. In particular, we can compute the outer hull of any discrete path (closed or not) as well as the intersection, union and difference of two Jordan discrete paths.

The next natural step would be to extend the ideas to closed discrete paths that are not necessarily Jordan curves, i.e. polyominoes having holes. In the same spirit, it would be interesting to check what complexity can be achieved when computing the overlay of multiple polyominoes instead of just two, as it appears here.

Acknowledgement

This work is an extension of results presented at the 18th international conference on Discrete Geometry for Computer Imagery (DGCI 2014) held in Siena, Italy from September 10th to 12th 2014 [6].

References

[1] A. Blondin Massé. *À l'intersection de la combinatoire des mots et de la géométrie discrète: Palindromes, symétries et pavages.* PhD thesis, Université du Québec à Montréal, February 2012.

[2] S. Brlek, M. Koskas, and X. Provençal. A linear time and space algorithm for detecting path intersection. *Theoretical Computer Science*, 412:4841–4850, 2011.

[3] S. Brlek, G. Labelle, and A. Lacasse. Algorithms for polyominoes based on the discrete green theorem. *Discrete Applied Mathematics*, 147(2-3):187–205, 2005.

[4] S. Brlek, G. Labelle, and A. Lacasse. A note on a result of daurat and nivat. In C. de Felice and A. Restivo, editors, *Developments in Language Theory, 9th International Conference, DLT 2005, Palermo, Italy, July 4-8, 2005, Proceedings*, volume 3572 of *Lecture Notes in Computer Science*, pages 189–198. Springer, 2005.

[5] S. Brlek, J.-O. Lachaud, X. Provençal, and C. Reutenauer. Lyndon + Christoffel = digitally convex. *Pattern Recognition*, 42:2239–2246, 2009.

[6] S. Brlek, H. Tremblay, J. Tremblay, and R. Weber. Efficient computation of the outer hull of a discrete path. In *Proceedings of the 18th IAPR International Conference on Discrete Geometry for Computer Imagery, DGCI 2014*, pages 122–133, Siena, Italy, 2014. Springer.

[7] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16:361–368, 1996.

[8] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete & Computational Geometry*, 10:377–409, 1993.

[9] A. Daurat and M. Nivat. Salient and reentrant points of discrete sets. *Discrete Applied Mathematics*, 151:106–121, 2005.

[10] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational geometry: Algorithms and applications.* Springer, third edition, 2008.

[11] Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[12] A. F. Filippov. An elementary proof of Jordan's theorem. *Uspekhi Mat. Nauk*, 5(39):173–176, 1950.

[13] H. Freeman. On the encoding of arbitrary geometric configurations. *IRE Transactions on Electronic Computers*, EC-10(2):260–268, 1961.

[14] J. E. Goodman and J. O'Rourke. *Handbook of discrete and computational geometry.* CRC Press, second edition, 2004.

[15] R. A. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[16] J. L. Gross and T. W. Tucker. *Topological graph theory.* Wiley, 1987.

[17] A. Blondin Massé, A. Garon, and S. Labbé. Combinatorial properties of double square tiles. *Theor. Comput. Sci.*, 502:98–117, 2013.

[18] X. Provençal. *Combinatoire des mots, géométrie discrète et pavages.* PhD thesis, Université du Québec à Montréal, September 2008.

[19] F. Spitzer. A combinatorial lemma and its application to probability theory. *Transactions of the American Mathematical Society*, 82:323–339, 1956.

[20] A. C.-C. Yao. *A lower bound to finding the convex hulls.* PhD thesis, Stanford University, April 1979.