RAIRO-Theor. Inf. Appl. 58 (2024) 16
https://doi.org/10.1051/ita/2024013

RAIRO - Theoretical Informatics and Applications
www.rairo-ita.org

ON THE GENERATION OF DISCRETE FIGURES WITH CONNECTIVITY CONSTRAINTS

Hugo Tremblay[1,*] and Julien Vernay[2]

Abstract. This paper addresses a generalization of polyominoes called (a,b)-*connected discrete figures*, where a and b represent the connectivity of the foreground (*i.e. black pixels*) and background (*i.e. white pixels*), respectively. Formally, a finite set of pixels P is (a,b)-connected if P is a-connected and $\overline{P}$ is b-connected. By adapting a combinatorial structure enumeration algorithm by J. L. Martin and employing breadth-first search ordering on the pixels of the figures, we sequentially generate all (a,b)-connected discrete figures up to size $n = 18$, utilizing minimal storage space. This paper presents an extended version of the research presented at the 2022 GASCom conference.

Mathematics Subject Classification. 05A15, 05B50.

Received December 13, 2022. Accepted July 11, 2024.

1. Introduction

Since their introduction by S. Golomb in [1], *Polyominoes*, that is finite sets of connected pixels, have proven to be of particular interest with many applications ranging from crystallography [2], robotics [3] and signal processing [4], among others. They appear in the literature under various monikers: animals [5, 6], clusters [7], polyominoes and pseudo-polyominoes [8] and self-avoiding polygons [9]. However we may call them, the problem of enumerating such discrete figures remains at the core of several research interests. This is not without reason as the efficient enumeration of polyominoes is still an unsolved problem.

In this paper, we concern ourselves with a generalization of polyominoes called (a,b)-*connected discrete figures*, where a and b respectively denotes the connectivity of the foreground (*i.e. black pixels*) and background (*i.e. white pixels*). Such objects were first introduced in [10] in order to study a digital image transformation problem. Formally, a discrete figure $P \subset \mathbb{Z}^2$ is (a,b)-connected if and only if P is a-connected and $\overline{P}$ is b-connected. We refer to 4-connected (resp. 8-connected) figures as $(4,0)$-connected (resp. $(8,0)$-connected) and we denote $S_{a,b}$ the set of (a,b)-connected discrete figures. It is easy to deduce that $S_{a,4} \subset S_{a,8} \subset S_{a,0}$ and $S_{4,b} \subset S_{8,b}$ for $a \in \{4,8\}$ and $b \in \{0,4,8\}$. Formal definitions are given in Section 2.

Some families $S_{a,b}$ have been considered in the literature: $S_{4,0}$ and $S_{8,0}$ are respectively polyominoes and pseudo-polyominoes [8]. Also, $(4,4)$-connected figures are frequently called *self-avoiding polygons* due to the fact that they can be generated by a self-avoiding walk on the grid $\mathbb{Z}^2$ [9]. To the best of our knowledge, $(4,8)$, $(8,4)$ and $(8,8)$-connected figures have never been previously studied.

Keywords and phrases: Combinatorics, digital geometry, polyominoes, exhaustive generation, algorithmics, parallelization.

[1] Department of Computer Science and Mathematics, Université du Québec à Chicoutimi, Canada.
[2] Asobo Studio, France.

[*] Corresponding author: hugo_tremblay2@uqac.ca

In [6], J. L. Martin introduces an algorithm for the enumeration of lattice graph structures by considering a canonical ordering on the neighbourhood of each vertex. This paper presents a modified version of this algorithm by considering the connectivity of both the foreground and the background of a polyomino in order to generate (a, b)-connected figures. Remark that a constant amortized time algorithm is given in [11], and further expanded in [12] for the generation of $(4, 4)$-connected polyominoes. However, our method does not make any assumptions about the type of connectivity of the figure and generates each figure without duplicates, thus using minimal memory space. Also, the computing time required to generate all figures of size n is proportional to the total number of figures, not to n itself. To the best of our knowledge, this paper constitutes the first enumeration of $(4, 8)$, $(8, 4)$ and $(8, 8)$-connected discrete figures. The enumeration of $(8, 0)$-connected figures up to size 18 is also a novel result. Finally, Martin's original algorithm was presented as a flowchart diagram and implemented in Fortran. We provide a more modern C++ implementation and a suitable pseudocode description.

2. Definitions and notations

Consider the *discrete grid* $\mathbb{Z}^2$. The *pixel* (or *cell*) $p(x, y)$ is the unit square $[x, y] \times [x + 1, y + 1]$, where (x, y) is a point of the discrete grid. Pixels are thus identified to $\mathbb{Z}^2$. Two pixels p and q are 4-*connected* (or 4-*adjacent*) if they have an edge in common and 8-*connected* (or 8-*adjacent*) if they have an edge or a vertex in common. The 4-*neighbourhood* of p is the set of all pixels 4-adjacent to p and is denoted by $N_4(p)$. We also define $N_4^*(p) = N_4(p) \cup \{p\}$. N_8 and N_8^* are defined in a similar manner.

A a-connected *(discrete) path* P is a sequence of pairwise a-connected pixels, that is

$$P = p_1, p_2, \ldots, p_n$$

where $p_{i+1} \in N_a(p)$. We then say that P is a path from p_1 to p_n of *length* n. Moreover, P is *closed* if $p_1 = p_n$. A set of pixels S is said to be locally a-connected if there exists in S a a-connected path between any pair of pixels of S. A (a, b)-*connected polyomino* (or (a, b)-*connected figure*) P is a finite set that is locally a-connected and such that the complement $\overline{P}$ (the infinite set of pixels not in P) is locally b-connected. The pixels of P are called the *black pixels* (or *foreground*) and those of $\overline{P}$ are called the *white pixels* (or *background*).

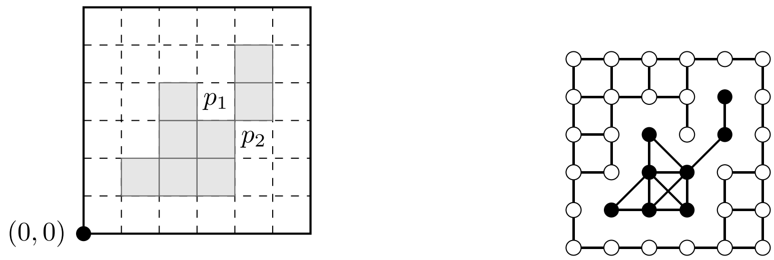
Let G_4 (respectively G_8) be the graph whose vertices are the set $\mathbb{Z}^2$ of pixels and where there is an edge between p and q if and only if they are 4-adjacent (respectively, 8-adjacent). Any polyomino is thus identified by a finite connected subgraph of G_4 (or G_8). A (a, b)-connected polyomino is identified to both a finite connected subgraph of G_a and an infinite connected subgraph of G_b. From a practical perspective, one cannot allow for G_b to be infinite in order to implement any sort of generation algorithm. To avoid this problem, we enclose the (a, b)-connected polyomino P in a suitably large rectangle such as $[x - 1, x' - 1] \times [y + 2, y' + 2]$ where x, x', y and y' are respectively the minimal x-coordinate of G_a, the maximal x-coordinate of G_a, minimal y-coordinate of G_a and maximal y-coordinate of G_a. Figure 1 gives an example of a (a, b)-connected polyomino and its associated graphs.

3. Generating (a, b)-connected figures

We now present a modified version of J. L. Martin's enumeration algorithm for generating (a, b)-connected discrete figures [6]. It proceeds by iteratively adding pixels to a figure until a prescribed size is attained. At each step, we ensure that both a and b connectivities for black and white pixels are maintained. This is done in constant time for black pixels since it suffices to select the next pixel among the neighbourhood of the pixels of P. The same approach also yields constant time methods for checking white connectivity for the cases $(4, 4)$, $(4, 8)$ and $(8, 4)$. However, checking white-connectivity for $(8, 8)$-figures is linear in the size of the figure since paths may intersect without having any pixel in common. These results are detailed in the following sections.

3.1. Black-connectivity

Maintaining black-connectivity can be done in constant-time by selecting pixels among a set of neighbours a-connected to the figure. This set is updated at each step by removing the pixel that was just added before



(A) The background of P is enclosed in the rectangle $[0, 0] \times [6, 6]$. Even though the white pixels p_1 and p_2 are 8-connected, the background is 4-connected since there exists a 4-connected path from p_1 to p_2.

(B) The graphs associated with P. The black pixels form a subgraph of G_8 and the white pixels form a subgraph of G_4.

FIGURE 1. A $(8, 4)$-connected polyomino P of size 8 and its corresponding graphs.

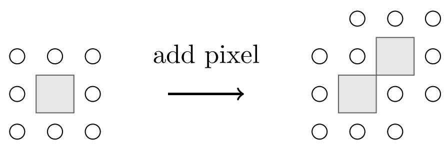


FIGURE 2. Adding a new pixel to a figure P adds, in the worst case, at most 4 white neighbours. This ensures the number of white pixels is linear in the size of P.

adding its a-neighbours to the set, if they are not already present. Because there is at most a neighbours added and that the choice of the pixel to add is done in constant time, the time complexity is $O(1)$, provided we choose a suitable data structure to store candidates. The data structure is further discussed in Section 4.2.

3.2. White-connectivity

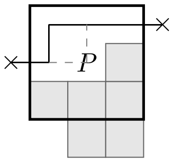
Checking whether adding a new pixel p to a figure breaks white-connectivity is more involved. We simplify the process by first checking whether the white neighbourhood of p (*i.e.* the white pixels 8-adjacent to p, also denoted $N_W(p)$) is b-connected. If it is indeed the case, then the white connectivity is maintained since the pixels of $N_W(p)$ are connected to the rest of the white pixels of $\mathbb{Z}^2$. Otherwise, we need to consider whether the figure has a hole and thus is not b-connected.

In order to check whether the white pixels are connected, we start by checking whether $N_W(p)$ is a connected subgraph of G_b. This has cost $O(1)$. Then, if $N_W(p)$ is not b-connected, we check whether B is connected, where B is the set of white pixels that are adjacent to a black pixel of the figure. This has cost $O(n)$ since there are at most $4(n + 1)$ white neighbours for a figure of size n (this follows from induction on the size of the figure and the fact that adding a new pixel adds at most 4 new white neighbours, see Fig. 2).

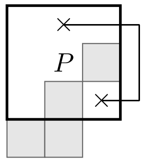
By restraining the connectivity to the cases $(4, 4)$, $(4, 8)$ and $(8, 4)$, it is sufficient to only check local connectivity. When adding a new pixel p to a figure, we consider the neighbourhood $N_W(p)$ consisting of the 8-adjacent neighbours of p. We then have one of the three cases depicted in Figure 3.

For case (a), $N_W(p)$ is a connected subgraph, meaning any white path which would potentially be disconnected by p becoming black remains connected *via* the local white neighbours of p.

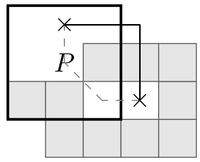
For case (b), $N_W(p)$ was not b-connected before the addition of p. This can only be the case for a white pixel diagonally adjacent to two black pixels, for white connectivity $b = 4$ (else, p would have linked all white neighbours). Since the previous figure is white-connected, there must exist paths between these white neighbours which do not include p, meaning these paths still exist and are not disconnected by the addition of p.



(A) $(a, b) = (4, 4)$

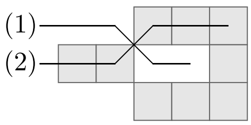


(B) $(a, b) = (8, 4)$

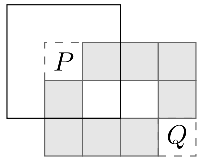


(C) $(a, b) = (4, 8)$

FIGURE 3. The three cases occurring while adding a new pixel p to the figure: (a) Local connectivity is preserved. Consequently, paths are redirected locally. (b) If local connectivity was already broken beforehand, then paths already exist between white components. (c) Local connectivity is broken by adding the pixel p. In this case, paths cannot be redirected since they would have to cross the figure.



(A) It is possible for $(8, 8)$-connected paths to intersect on a vertex of a pixel



(B) Considering $N(p)$ is not sufficient to determine (a, b)-connectivity since it depends on whether the pixel Q is black or white.

FIGURE 4. For $(8, 8)$-connected figures, it is not sufficient to study $N(p)$ in order to determine the connectivity of the figure obtained from the addition of p.

For case (c), the addition of p disconnects the white pixels of the figure P, which partitions the white pixels into two sets: white pixels interior to P and white pixels exterior to P. For connectivities other than $(8, 8)$, any path which connects an interior white pixel to an exterior white pixel must cross the figure, meaning the figure is not b-connected anymore.

Consequently, ensuring the $(4, 4)$, $(4, 8)$ and $(8, 4)$-connectivity is done by checking the previous three cases for the neighbourhood of p, yielding a constant-time method for adding new pixels. The check implementation is explained in Section 4.4. However, this approach does not work for $(8, 8)$-connectivity since it is possible for two paths to cross without intersecting on a pixel. Figure 4 illustrates this particular case. Thus, a graph traversal is required for $(8, 8)$-connectivity.

3.3. Algorithm for generating (a, b)-connected polyominoes

Before presenting our algorithm for generating (a, b)-connected polyominoes, we offer a brief overview of J.L. Martin's algorithm. First, remark that adding pixels one at a time defines an ordering on the pixels of P. Since this ordering is not unique (*e.g.* there are four ways of constructing the 2 by 2 square polyomino, starting at the bottom left pixel), this process may produce duplicate figures. The idea behind Martin's algorithm is to instead explicitly define a unique ordering on the pixels of P. Consequently, a figure is constructed by considering the so-called *canonical ordering* of its pixels. By defining a first pixel p (*e.g.* the left-most pixel on the bottom row) and a visiting order of the neighbours of p (*e.g.* anticlockwise starting with the pixel to the right of p), the canonical ordering of the pixels of a figure corresponds to the breadth-first traversal of the graph G_4 (or G_8). An example is given in Figure 5.

Martin's algorithm uses this ordering to define "add" and "remove" operations on the current set of pixels. Figures are then constructed in a specific order in relation to this canonical ordering. Additionally, it specifies

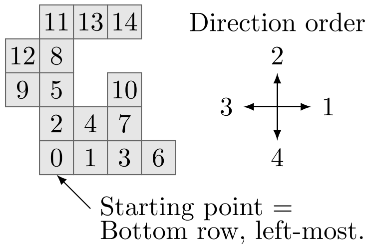


FIGURE 5. Example of canonical ordering using the left-most point on the bottom row as a starting pixel along with the cyclic counterclockwise ordering right-up-left-down.

"prohibited" elements as to not generate an already visited figure. It is worth noting that Martin did not provide pseudocode, implementation details or data structures for his algorithm, relying instead on a simple flowchart description of the procedure.

We now propose a new algorithm for generating (a, b)-connected polyominoes by re-imagining Martin's algorithm as a tree traversal where nodes contain polyominoes and by considering arbitrary connectivity constraints such as white connectivity. To be more precise, for each figure P of size n, its parents are the valid figures of size $n-1$ obtained by removing a pixel from P. We define the *unique canonical parent of* P as the figure obtained by removing from P the last added pixel in the canonical ordering. Conversely, a *canonical child of* P is a figure whose canonical parent is P. With this relationship, the set of all figures can be viewed as a tree structure, called *canonical tree*, whose root is the sole figure of size 1. Our algorithm visits this tree using a depth-first traversal, generating new figures one at a time while keeping track of "prohibited" pixels so as not to generate non-canonical children figures.

Formally, our algorithm requires an ordered set of pixels, called *candidate pixels*, and a state for the current figure. The candidate pixels are all pixels chosen for the current figure together with all neighbour pixels which may either be "free" (*i.e.* may be added to the current figure to generate a new figure) or "prohibited" (*i.e.* adding them would generate a previously generated figure).

At first, the set of candidates is initialized by choosing a so-called origin pixel serving as the root of the tree. This defines the starting figure as the sole figure of size 1. We then define the three primitive operations `firstChild()`, `nextSibling()` and `parent()`, detailed below.

- `firstChild()`: adds the neighbours of the last chosen pixel as free candidates. Then, the next free pixel p among the set of candidates is added to the current figure.
- `nextSibling()`: takes the next free pixel p among the set of candidates. If p does not exist, there are no more siblings. Else, the last added pixel is removed and marked as prohibited. Then, p is added.
- `parent()`: reverts the last chosen candidate state as "free" before removing all pixels added by the corresponding `firstChild()` operation. Finally, all prohibited pixels after the last chosen candidate in the ordering are marked as "free" candidates. In doing so, pixels are marked as prohibited only during the previous sub-tree.

After applying either `firstChild()` or `nextSibling()`, a new possible figure is obtained. The various connectivity constraints are then checked (*e.g.* white connectivity). If they are satisfied, then the new figure is added. Otherwise, `nextSibling()` is called, effectively skipping this invalid figure and its sub-tree.

While adding candidates, care must be taken to not include pixels which would contradict the starting point criteria. For instance, if the starting point is the left-most pixel on the bottom row, pixels on rows below the origin pixel cannot be candidates, nor pixels to the left of the origin pixel. These contradictory pixels can be easily dealt with by marking them as "prohibited" during the initialization. This way, they stay prohibited during the entire generation procedure.

The pseudocode, data structure and complexity analysis for our algorithm are discussed in Section 4. Figure 6 depicts the result of applying our algorithm for $(4, b)$-connected figures of size $n \leq 4$ where $b \in \{0, 4, 8\}$.

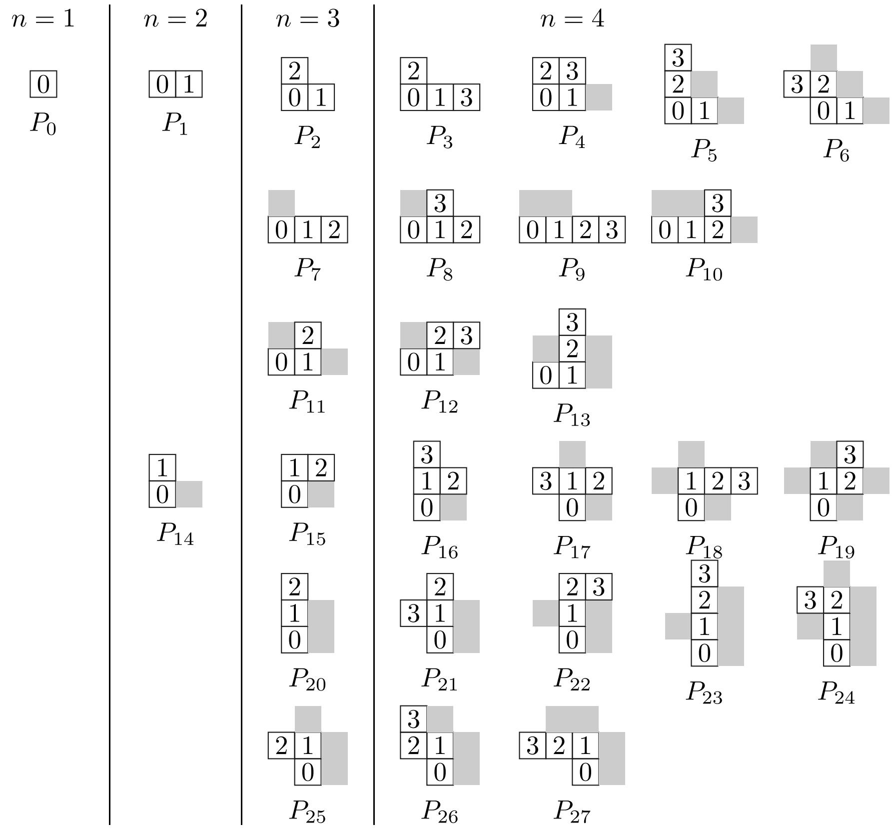


FIGURE 6. All 28 figures with $n \leq 4$ and black connectivity $a = 4$, generated by our algorithm. The number on each pixel corresponds to the pixel's position in the canonical ordering as in Figure 5. Each figure's canonical parent is the previous figure of size $n - 1$. Grey pixels are prohibited; using them would result in an already visited figure of size $n + 1$. For instance, adding P_{11}'s top-left pixel would give P_4.

4. Implementation

S. Redner proposed in 1982 a FORTRAN implementation of Martin's algorithm for the generation of 4-connected figures without considering white connectivity [7]. We propose a thoroughly documented and modern C++ implementation of our algorithm, freely available on GitHub [13].

4.1. Grid, pixels and directions

First, we bound the infinite 2D plane containing the figures, as mentioned in Section 2. We choose `Nmax` as an arbitrary limit for the size of the figure. Since the starting point is defined to be the leftmost lowest pixel, the figure only grows in the three directions north, east and west, up to a distance of $\texttt{Nmax} - 1$ pixels. This ensures all generated figures fit on a grid of $(2 \cdot \texttt{Nmax} - 1) \times (\texttt{Nmax})$ pixels, with the starting point in the middle of the first row. In our implementation, we add a margin of two pixels in each direction so the figures are generated in a grid of $\texttt{Width} = 2 \cdot \texttt{Nmax} + 3$ and $\texttt{Height} = \texttt{Nmax} + 4$. This way, each white pixel has access to its neighbourhood, regardless of connectivity.

The grid is considered one-dimensional, that is each pixel $p(x, y)$ is mapped to the integer $\texttt{pos} = x + y \times \texttt{Width}$, $\texttt{pos} \in [0, \texttt{Width} \times \texttt{Height}[$, which denotes the position of the pixel in the grid. This allows

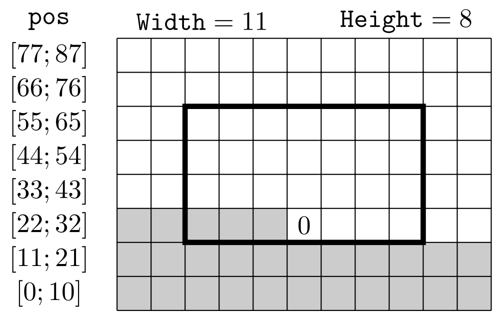


FIGURE 7. Grid representation for $\texttt{Nmax} = 4$. The pixel labelled 0 is the starting point at position $\texttt{PosOrigin} = 27$. Grey pixels are prohibited from the start: adding them would break the starting point criteria. The black rectangle represents the area where the figure is generated.

the use of a pre-allocated array for storing the pixels. Accessing a particular pixel is done in $O(1)$ and neighbouring pixels are accessed by simply adding an offset: right, up, left and down neighbours of pixel at `pos` have index $\texttt{pos} + 1$, $\texttt{pos} + \texttt{Width}$, $\texttt{pos} - 1$ and $\texttt{pos} - \texttt{Height}$ respectively, using the convention that the bottom-left pixel is $p(0, 0)$.

Taking into account the two pixels of margin, the starting pixel, which is in the middle of the first row, is $p(\lfloor \texttt{Width} \div 2 \rfloor, 2)$ and has position $\texttt{PosOrigin} = \lfloor \texttt{Width} \div 2 \rfloor + 2 \cdot \texttt{Width}$. In order to respect the starting point criteria, one must not use the pixels to the left or on lower rows as the starting point. In the grid representation, these forbidden pixels are those such that $\texttt{pos} < \texttt{PosOrigin}$. See Figure 7 for a visual representation of the grid.

4.2. Data structures and state

In order to represent the ordered set of candidates, we use an array of integers `candidates` (the candidates' positions) along with an integer `count` (the number of candidates) and a grid of $\texttt{Width} \times \texttt{Height}$ bits `gridCandidates` (whether a pixel is already in the set). The array `candidates` is pre-allocated, as the total number of candidates is bounded by $5 \cdot \texttt{Nmax}$ (the worst-case being a diagonal figure in G_8, where each chosen pixel adds 5 new candidates as in Fig. 2). When adding a pixel to the set, we verify that $\texttt{gridCandidates}[\texttt{pos}]$ is zero, in which case, `pos` is assigned to $\texttt{candidates}[\texttt{count}]$, then `count` is incremented. Removing pixels is always done from last to first: for pixel at $\texttt{pos} = \texttt{candidates}[\texttt{count} - 1]$, we reset the bit $\texttt{gridCandidates}[\texttt{pos}]$ to zero, and decrement `count`. Consequently, both adding and removing pixels have complexity $O(1)$.

We represent the current depth of iteration as an integer `level`, which denotes the canonical ordering of the pixel we are currently modifying (starting at index 0), giving figures of size $n = \texttt{level} + 1$. The current figure is stored in an array of `Nmax` integers `chosenIndices`, containing the index of each chosen candidate in `candidates`. This ensures the position of the last chosen pixel is $\texttt{candidates}[\texttt{chosenIndices}[\texttt{level}]]$. Additionally, since white connectivity checks require fast access to whether a pixel at `pos` is chosen or not, we also maintain an array of bits `gridChosen` (whether a pixel is chosen) indexed by `pos`.

Finally, we need to recall which candidates where added by the function `firstChild()` so that the function `parent()` may remove them. This is done with an array of `Nmax` integers `candidatesCounts`, containing the values of `count` after the `level`-th `firstChild()` operation. During the `parent()` operation, the candidates to be removed are $\texttt{candidates}[i]$ for $i \in [\texttt{candidatesCounts}[\texttt{level}], \texttt{count}[$. Then, $\texttt{candidatesCounts}[\texttt{level}]$ is assigned back to `count`, and `level` is decremented.

In our implementation, we do not explicitly mark whether a candidate is free or prohibited. This state can be retrieved from the values of `chosenIndices`. For each candidate at $\texttt{candidates}[\texttt{idx}]$, its *state* is:

- "chosen" if $\texttt{chosenIndices}[k] == \texttt{idx}$ with $k \leq \texttt{level}$,
- "free" if $\texttt{idx} > \texttt{chosenIndices}[\texttt{level}]$, and

- "prohibited" if $\texttt{chosenIndices}[k] < \texttt{idx} < \texttt{chosenIndices}[k+1]$ with $k < \texttt{level}$. The pixel stays prohibited until we call `parent()` when $\texttt{level} = k+1$.

With this representation, the next "free" pixel exists if $\texttt{chosenIndices}[\texttt{level}] + 1 < \texttt{count}$, in which case its position is $\texttt{candidates}[\texttt{chosenIndices}[\texttt{level}] + 1]$. A pixel is automatically marked as "prohibited" whenever a pixel further in the `candidates` ordering is "chosen". The pixel reverts to the "free" state whenever all pixels after it in the `candidates` ordering are "un-chosen".

4.3. Pseudocode and complexity

Algorithm 1 `firstChild()` primitive

```
idx ← chosenIndices[level]
for pos ∈ N_a(candidates[idx]) do                  ▷ Add last chosen pixel's neighbours
    if gridCandidates[pos] = 0 then
        gridCandidates[pos] ← 1
        candidates[count] ← pos
        count ← count + 1
    end if
end for
if idx + 1 ≥ count then                            ▷ If no candidates are available
    return FAIL
end if
level ← level + 1
candidatesCounts[level] ← count
chosenIndices[level] ← idx + 1
gridChosen[candidates[idx + 1]] ← 1
return SUCCESS
```

Algorithm 2 `nextSibling()` primitive

```
idx ← chosenIndices[level]
if idx + 1 < count then
    gridChosen[candidates[idx]] ← 0
    gridChosen[candidates[idx + 1]] ← 1
    chosenIndices[level] ← idx + 1
    return SUCCESS
else
    return FAIL
end if
```

Algorithm 3 `parent()` primitive

```
gridChosen[candidates[chosenIndices[level]]] ← 0
level ← level − 1
for idx ∈ [candidatesCounts[level]; count[ do
    gridCandidates[candidates[idx]] ← 0
end for
count ← candidatesCounts[level]
```

Algorithm 4 Algorithm to generate (a, b)-connected figures

```
candidates[0] ← PosOrigin
count ← 1
candidatesCounts[0] ← 1
for pos ∈ [0; PosOrigin] do                          ▷ Prevent adding contradictory pixels
    gridCandidates[pos] ← 1
end for
chosenIndices[0] ← 0
gridChosen[PosOrigin] ← 1
level ← 0

while not done do                                    ▷ Main loop (A)
    while Current figure valid do                    ▷ Go deeper loop (B)
        Yield current figure
        if level ≥ Nmax − 1 then
            break
        end if
        if firstChild() fails then
            break
        end if
    end while
    while nextSibling() fails do                     ▷ Next sub-tree loop (C)
        if level = 0 then                            ▷ Termination condition
            return DONE
        end if
        parent()
    end while
end while
```

Our method for generating (a, b)-connected figures is detailed in Algorithm 4. It relies on the three visiting procedures `firstChild()`, `nextSibling()` and `parent()`, defined respectively in Algorithms 1, 2 and 3. `nextSibling()` may fail if the last chosen pixel is the last of the candidates, meaning there is no free candidate available. `firstChild()` may fail too, either if current figure has maximum size, or if the last chosen pixel is the last candidate and its neighbours are already candidates.

In order to analyze the complexity of our algorithm, we start with Algorithms 1, 2 and 3. Since arrays are contiguous and they only make use of simple integer arithmetic, they are accessed in constant-time, making each individual instruction of complexity $O(1)$. The loop inside `firstChild()` visits the neighbours of the last added pixel, of which there are a, with $a \in \{4, 8\}$, (*i.e.* the black connectivity). In particular, the number of iteration does not depend on the size of the figures, so this loop has complexity $O(1)$. The loop inside `parent()` removes candidates added inside the loop of the last `firstChild()`. There are at most a candidates added by `firstChild()`, so there are at most a candidates removed by `parent()`, meaning the loop inside `parent()` has complexity $O(1)$. Therefore, Algorithms 1, 2 and 3 each have complexity $O(1)$.

To further our complexity analysis, we define disjoint classes of figures and their total number. The union of these classes gives the set of figures of size $n \geq$ `Nmax`, with black connectivity a, without considering the white connectivity constraint.

- **#NonLeaf**: the number of valid figures which have canonical children left to visit.
- **#Leaf**: the number of valid figures which do not have canonical children left to visit: these include the figures with maximum size $n =$ `Nmax`.

TABLE 1. Algorithm 4 time complexity, for all types of connectivity.

Connectivity	Total complexity for all figures	Amortized complexity for one figure
$(4,0)$, $(8,0)$	$O(\#\textbf{Valid})$	$O(1)$
$(4,4)$, $(4,8)$, $(8,4)$	$O(\#\textbf{Valid} + \#\textbf{Rejected})$	$O(1)$
(8,8)	$O(n \times (\#\textbf{Valid} + \#\textbf{Rejected}))$	$O(n)$

- #**Valid**: the sum of #**NonLeaf** and #**Leaf**.
- #**Rejected**: the number of invalid figures, where their canonical parents are valid. These are the figures rejected by the validity check. This is zero when we do not check white connectivity.
- #**Skipped**: the number of invalid figures, where their canonical parents are themselves invalid. These figures are irrelevant to our algorithm: They are not explored since their invalid canonical parent is already rejected.

Algorithm 4 consists of a main loop (A), itself containing two loops (B) and (C) comprised of the validity check procedure and the visit primitives procedure respectively. We now count the total number of these operations:

- `firstChild()` is called once per each valid figure: #**Valid** times. It succeeds #**NonLeaf** times and fails #**Leaf** times.
- `parent()`: Algorithm 4 terminates when `level` $= 0$ is reached for a second time. `level` is only incremented on successful `firstChild()` calls and only decremented on `parent()`. Consequently, `parent()` is called as many times as successful `firstChild()` calls, that is #**NonLeaf** times.
- `nextSibling()` is called once after each exit of loop (B), and once after each call to `parent()` (#**NonLeaf** times). Loop (B) is exited after either `firstChild()` fails (#**Leaf** times) or a figure is rejected (#**Rejected** times).
- Validity check is done once per each valid figure and each rejected figure, that is #**Valid** + #**Rejected** times.

All operations are called at most #**Valid** + #**Rejected** times. The complexity of the visit primitives procedure is $O(1)$, and the validity check procedure has complexity $O(1)$ or $O(n)$, as mentioned in Section 3.2. The total time complexity of Algorithm 4 for generating all (a, b)-connected figures is then the complexity of the validity check procedure times #**Valid** + #**Rejected**. The amortized time complexity for generating the next figure is the total complexity divided by the number #**Valid** of generated figures. Remark that, since #**Rejected** $\leq$ #**Valid**, we have $O\left(n + \frac{\#\textbf{Rejected}}{\#\textbf{Valid}}\right) = O(1)$ and $O\left(n + n\frac{\#\textbf{Rejected}}{\#\textbf{Valid}}\right) = O(n)$. The time complexity for the different connectivity are summed up in Table 1, where connectivity $(a, 0)$ means that white connectivity check is disabled.

The memory usage has complexity $O(n^2)$ because we store two-dimensional grids. This is not an issue in practice since, in our implementation, the various states defined in Section 4.2 consume, in total, at most 1344 bytes for generating figures with $n \leq 20$. Remark that it would require at most 11416 bytes for sizes $n \leq 100$ (the worst case in our implementation being $(8, 8)$-connected figures). Table 2 gives the values of #**Valid** and #**Rejected** with `Nmax` $= 13$, and the memory size required by our algorithm.

4.4. Efficient white-connectivity check

The white-connectivity check is applied to all valid and rejected figures, so its performance is paramount for fast generation of figures. As discussed in Section 3.2, validity check for connectivity $(4, 4)$, $(4, 8)$ and $(8, 4)$ is done by only considering the white neighbours of the last chosen candidate $N_W(p)$. In the following discussion, we call A, B, C, D, F, G, H, I the eight neighbours of p, as shown in Figure 8. We refer to them as "white" if they

TABLE 2. Values of #**Valid** and #**Rejected** for figures of size $n \leq 13$, along with the memory size required by the algorithm.

Connectivity	#**Valid**	#**Rejected**	$\frac{\#\textbf{Rejected}}{\#\textbf{Valid}}$(%)	Memory consumption
$(4,0)$	2 595 167	0	0	344 bytes
$(4,8)$	2 577 792	4679	0.18%	664 bytes
$(4,4)$	2 377 414	69 730	2.93%	664 bytes
$(8,0)$	1 996 505 920	0	0	344 bytes
$(8,8)$	1 996 369 432	24 770	<0.01%	920 bytes
$(8,4)$	1 645 507 929	28 350 917	1.72%	664 bytes

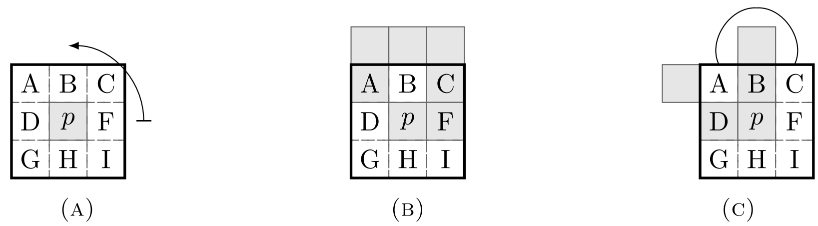


FIGURE 8. (a) Names of the neighbours. (b) Example where a (4,8)-connected figure would report broken white-connectivity due to lack of diagonal handling. (c) Example where a (8,4)-connected figure would report broken but it was already previously broken locally.

are not chosen in the figure. We count the number of white components $c_{(a,b)}$ in $N_W(p)$, relying on Boolean arithmetics instead of graph algorithms. By considering the neighbours of p as Booleans with `TRUE` $= 1$ if chosen and `FALSE` $= 0$ if not, white connectivity is preserved if $c_{(a,b)} \leq 1$.

First, for connectivity $(4,4)$, the number of white connected sets can be found as the number of adjacent pairs (p_1, p_2) in the loop $F \to C \to B \to \cdots \to I \to F$ where p_1 is black and p_2 is white.

$$\begin{aligned} c_{(4,4)} = \quad & (F \texttt{ AND } \overline{C}) + (C \texttt{ AND } \overline{B}) + (B \texttt{ AND } \overline{A}) + (A \texttt{ AND } \overline{D}) \\ & + (D \texttt{ AND } \overline{G}) + (G \texttt{ AND } \overline{H}) + (H \texttt{ AND } \overline{I}) + (I \texttt{ AND } \overline{F}) \end{aligned} \tag{4.1}$$

For connectivity $(4,8)$, the same computation can be reused, but the count has false positives when one of $\{A, C, G, I\}$ is black and its two adjacent pixels are white. Figure 8b shows such a false positive example with pixels A, B, D. These cases must be decremented from the count.

$$\begin{aligned} c_{(4,8)} = c_{(4,4)} & - (A \texttt{ AND } \overline{B} \texttt{ AND } \overline{D}) - (C \texttt{ AND } \overline{B} \texttt{ AND } \overline{F}) \\ & - (G \texttt{ AND } \overline{D} \texttt{ AND } \overline{H}) - (I \texttt{ AND } \overline{F} \texttt{ AND } \overline{H}) \end{aligned} \tag{4.2}$$

For connectivity $(8,4)$, we reuse the computation for $(4,4)$, but there may be two white components legitimately, when one of $\{A, C, G, I\}$ is white and its two adjacent pixels are black. In this situation, the white component in the corner is still connected with an external path to the rest of the white neighbours, as shown in Figure 8c : A and the center p were both white before p being chosen, but they are not connected locally. These cases must be removed from c.

$$\begin{aligned} c_{(8,4)} = c_{(4,4)} & - (\overline{A} \texttt{ AND } B \texttt{ AND } D) - (\overline{C} \texttt{ AND } B \texttt{ AND } F) \\ & - (\overline{G} \texttt{ AND } D \texttt{ AND } H) - (\overline{I} \texttt{ AND } F \texttt{ AND } H) \end{aligned} \tag{4.3}$$

TABLE 3. Performance result of our multi-threaded implementation of Algorithm 4.

Parameters	Figures count	Total time	Figures per second
$(4,0), \texttt{Nmax} = 20$	30 988 922 366	14.2 s	2167.7 millions
$(4,8), \texttt{Nmax} = 20$	30 334 771 986	38.9 s	779.9 millions
$(4,4), \texttt{Nmax} = 20$	24 681 869 833	33.4 s	738.9 millions
$(8,0), \texttt{Nmax} = 15$	87 776 030 494	38.6 s	2270.4 millions
$(8,8), \texttt{Nmax} = 15$	87 767 553 560	321.0 s	273.4 millions
$(8,4), \texttt{Nmax} = 15$	69 192 311 923	83.1 s	832.6 millions

Finally, we represent those eight Booleans using a single integer by using one bit per Boolean. This integer is used as an index in a precomputed lookup table of 256 entries. Whenever we want to check if choosing a candidate would break white-connectivity, we simply access this table instead of doing the computation for every enumerated figure.

4.5. Parallelization of the algorithm

Algorithm 4 is equivalent to a depth-first traversal of the imaginary "canonical tree" of figures, until reaching depth `Nmax`. All children figures in this tree have a single parent, so that figures of size n only have one ancestor of a given size $n' < n$. Due to this property, our algorithm is easily separated to perform independent tasks.

First, we choose a small n'. We generate all figures of size $n \leq n'$ and we store our algorithm's state for each figure of size $n = n'$. After this short single-threaded pass, we get multiple copies of our state, which can be resumed independently without limiting depth to n'. We then create one task per state copy, which starts the iteration again, limiting the depth to `Nmax` and terminating when we find a figure of size $n \leq n'$. At the end, we aggregate the result of all these passes to get the figures' count.

In practice, we choose $n' = 8$ or $n' = 6$, respectively for black connectivities $a = 4$ and $a = 8$, to get a few thousand independent tasks. As stated at the end of Section 4.3, our algorithm's state requires about one kilobyte of memory space, so storing a few thousand copies consume a few megabytes of memory. This parallelization could theoretically run on multiple computers. However, we do not have access to the required infrastructure. Instead, our implementation uses a C++17 thread-pool library made by Barak Shoshany [14].

4.6. Results and performances

We implemented Algorithm 4 in both single-thread and multi-thread versions for all types of connectivity. For a given size n, we measured both the number of generated figures as well as the execution time. We used a laptop with Windows 11 and an Intel® Core i7-13700HX (24 logical cores). The results are in Table 3. We distinguish three speed categories:

- Connectivity $(4,0)$ and $(8,0)$ (without white connectivity checks) are the fastest, with about 2200 million figures generated per second. Extra speed was achieved by not maintaining `gridChosen`, which is only necessary for white connectivity check.
- Connectivity $(4,4)$, $(4,8)$ and $(4,4)$ allows for the generation of about 800 million figures per second. The extra time is due to maintaining `gridChosen` and the lookup access for white connectivity check.
- Connectivity $(8,8)$ is the worse, due to the $O(n)$ validity check.

We implemented Algorithm 4 in C++. The results for the number of figures of each connectivity type are in Tables 4 and 5. Also, Figure 9 presents minimal examples explaining the difference in cardinality of the various families $S_{a,b}$.

TABLE 4. Number of (a, b)-connected figures. The results for $(4, 0)$ and $(4, 4)$ respectively correspond to OEIS sequence #A001168 for the number of fixed 4-connected polyominoes and OEIS sequence #A006724 for the number of self-avoiding polygons.

n	$(4, 0)$-connected	$(4, 8)$-connected	$(4, 4)$-connected
1	1	1	1
2	2	2	2
3	6	6	6
4	19	19	19
5	63	63	63
6	216	216	216
7	760	760	756
8	2725	2724	2684
9	9910	9898	9638
10	36 446	36 358	34 930
11	135 268	134 744	127 560
12	505 861	503 065	468 837
13	1 903 890	1 889 936	1 732 702
14	7 204 874	7 138 286	6 434 322
15	27 394 666	27 086 832	23 993 874
16	104 592 937	103 202 581	89 805 691
17	400 795 844	394 625 770	337 237 337
18	1 540 820 542	1 513 810 138	1 270 123 530
19	5 940 738 676	5 823 764 372	4 796 310 672
20	22 964 779 660	22 462 566 215	18 155 586 993
Total	30 988 922 366	30 334 771 986	24 681 869 833
Time	14.2 s	38.9 s	33.4 s
Speed	2167.7 $\mathrm{M\,s^{-1}}$	779.9 $\mathrm{M\,s^{-1}}$	738.9 $\mathrm{M\,s^{-1}}$

5. Conclusion

This paper deals with (a, b)-connected discrete figures, that is finite sets of a-connected pixels such that the background is b-connected. By modifying Martin's enumeration algorithm, we generate all such figures as well as their number up to size 18 for most families $S_{a,b}$. We also discuss proofs for the complexity of checking black and white connectivity at each step.

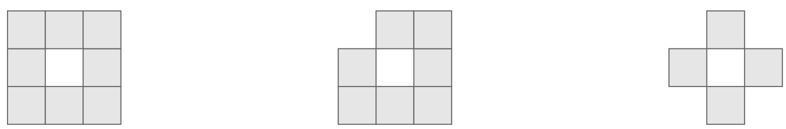
This work constitutes a first step towards understanding this type of combinatorial objects and opens up several new interesting research avenues. From a theoretical point of view, it would be interesting to find generating series for (a, b)-connected figures. We could also hope to find asymptotics bounds for this type of figures. For instance, methods developed in [15] could be useful in providing additional insight into (a, b)-connected figures.

Our algorithm is easily applicable to regular tilings (hexagonal, triangular, rectangular) in $\mathbb{R}^n$. For instance, Algorithm 4 is valid for the 3D cases, provided a suitable connectivity test for white pixels is defined.

Also, since they originated from a digital picture transformation problem in [10], the work done in this paper could help solve the following conjecture: Given two $(4, 4)$-connected digital images I and I' of size n, there exists a sequence of $O(n^2)$ 8-adjacent pixel interchanges that transforms I into I'. A possible starting point for tackling this problem would be to study the density of (a, b)-connected discrete figures and its correlation to the preceding transformation. Recall that the *density* of a simple graph G on n vertices, and by extension a discrete figure, effectively measures how close G is to the complete graph K_n. It is relatively straightforward to show that the density tends to zero as n grows.

TABLE 5. Number of (a, b)-connected figures. The result for $(8, 0)$ corresponds to OEIS sequence #A006770 for the number of fixed 8-connected polyominoes, extended here to $n = 18$ for the first time. Generating $(8, 8)$-connected figures of size $n \geq 18$ would have taken too much time on our computer.

n	$(8,0)$-connected	$(8,8)$-connected	$(8,4)$-connected
1	1	1	1
2	4	4	4
3	20	20	20
4	110	110	109
5	638	638	622
6	3832	3832	3664
7	23 592	23 592	22 094
8	147 941	147 940	135 609
9	940 982	940 966	843 941
10	6 053 180	6 053 002	5 310 754
11	39 299 408	39 297 724	33 724 862
12	257 105 146	257 090 547	215 793 158
13	1 692 931 066	1 692 811 056	1 389 673 091
14	11 208 974 860	11 208 021 976	8 998 648 488
15	74 570 549 714	74 563 162 152	58 548 155 506
16	498 174 818 986	498 118 512 909	382 526 638 033
17	3 340 366 308 393	3 339 942 522 834	2 508 473 632 910
18	22 471 158 811 164	N/A	16 503 616 943 998
Total	26 397 475 969 037	3 925 828 589 303	19 463 809 526 864
Time	3 h 7 min 53.9 s	4 h 8 min 58.7 s	6 h 41 min 26.5 s
Speed	2341.5 $\mathrm{M\,s^{-1}}$	262.8 $\mathrm{M\,s^{-1}}$	808.1 $\mathrm{M\,s^{-1}}$



(A) $P \in S_{4,0}$ but $P \notin S_{4,8}$ (B) $P \in S_{4,8}$ but $P \notin S_{4,4}$ (C) $P \in S_{8,8}$ but $P \notin S_{8,4}$

FIGURE 9. Examples of discrete figures explaining the difference in number between the families $S_{a,b}$.

The random generation of (a, b)-connected polyominoes is also an interesting question. A Monte-Carlo version of Martin's algorithm was proposed by P. M. Lam in 1986 [5] where each generated figure has a non zero probability of being discarded. We briefly explored a generalization of this algorithm by randomly selecting the next generated figure among a set of valid candidates. Even though these methods do not produce figures uniformly at random, this is a worthwhile research avenue to pursue.

From an experimental point of view, it would be interesting to refine our implementation, namely by using a more powerful computer or by using a bigger network of computers for more efficient parallelization. Experimenting with various segmentation values or probabilities would also surely lead to interesting results, both for the random and the exhaustive generation of discrete figures. In particular, the computation of the number of $(8, 8)$-connected discrete figures of size 18 could be easily and quickly obtained by using a more powerful computer.

References

[1] S.W. Golomb, Polyominoes. Charles Scribners' Sons (1965).

[2] T. Sunada, Topological Crystallography. Springer (2013).

[3] M. Kobilarov, M. Desbrun, J. Marsden and G. Sukhatme, A discrete geometric optimal control framework for systems with Symmetries, in *Proceedings of Robotics: Science and Systems*, Atlanta, GA, USA (2007).

[4] Z.-M. Du, F.-Y. Ye, H. Shi and G.-P. Zhu, A fast recovery method of 2D geometric compressed sensing signal. *Circuits Syst. Signal Process.* **34** (2015) 1711–1724.

[5] P.M. Lam, On Monte Carlo generation and study of anisotropy of lattice animals. *J. Phys. A: Math. Gen.* **19** (1986) L155.

[6] J.L. Martin, Phase Transitions and Critical Phenomena, Vol. 97. Academic Press, London and New York (1974) 97–112.

[7] S. Redner, A Fortran program for cluster enumeration. *J. Stat. Phys.* **29** (1982) 309–315.

[8] S.W. Golomb, Polyominoes: Puzzles, Patterns, Problems, and Packings. Princeton University Press (1994).

[9] I. Jensen and A.J. Guttmann., Self-avoiding polygons on the square lattice. *J. Phys. A: Math. Gen.* **32** (1999) 4867.

[10] P. Bose, V. Dujmovic, F. Hurtado and P. Morin, Connectivity-preserving transformations of binary images. *Comput. Vis. Image Understand.* **113** (2009) 1027–1038.

[11] P. Massazza, Hole-free partially directed animals, in *Developments in Language Theory*, edited by P. Hofman and M. Skrzypczak. Springer International Publishing, Cham (2019) 221–233.

[12] V. Dorigatti and P. Massazza, Partially directed animals with a bounded number of holes, in *Language and Automata Theory and Applications*, edited by A. Leporati, C. Martín-Vide, D. Shapira and C. Zandron. Springer International Publishing, Cham (2021) 16–28.

[13] J. Vernay, Github repository: discrete-figures (2023).

[14] B. Shoshany, A c++17 thread pool for high-performance scientific computing. (2021).

[15] P. Flajolet and R. Sedgewick, Analytic Combinatorics, 1st edn. Cambridge University Press, USA (2009).