INF421 PROGRAMMING PROJECT
POLYOMINO TILINGS AND EXACT COVER

VINCENT PILAUD
vincent.pilaud@lix.polytechnique.fr

A *polyomino* is set of squares in the plane that are connected along edges. Dually, it is a finite connected induced subgraph of the integer grid. Given a set S of polyominoes and a polyomino P, a *polyomino tiling* of P with S is a partition of P by the polyominoes of S. See Figure 1. The goal of this project is to answer as efficiently as possible questions like:

- generate all polyominoes of a given area up to isometries,
- find all polyomino tilings of P with S,
- find all possible polyomino tilings of a rectangle by all polyominoes of a given area,
- find dilates of P that can be tiled by copies of P, etc.

In general, polyomino tiling problems are known to be computationally difficult. This project aims at implementing an efficient backtracking technique for these problems.

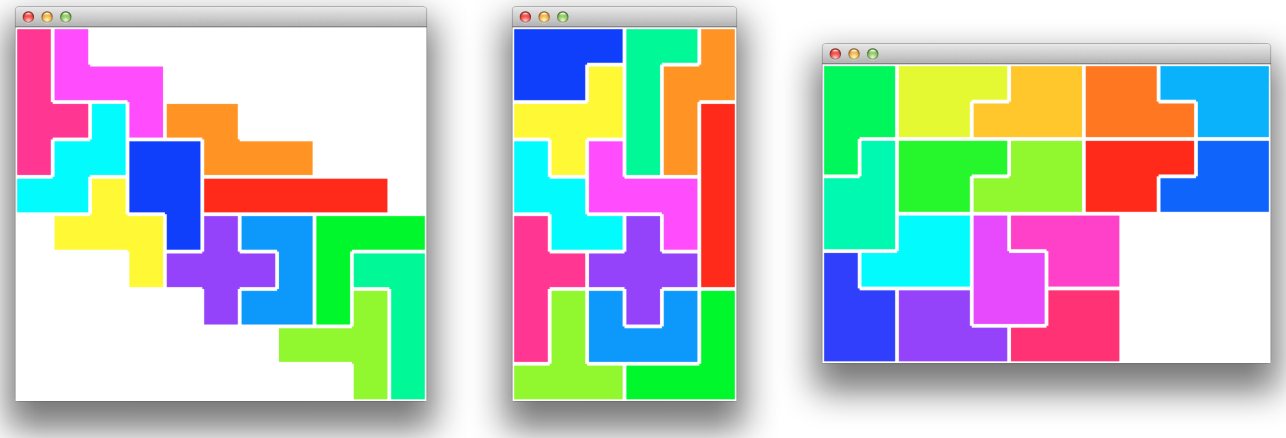

FIGURE 1. Three polyomino tilings: in an arbitrary polyomino (left), in a rectangle (middle), and in a dilate of the pieces (right).

The expected programming language for this project is Java, and the material is prepared in Java. However, other programming languages are accepted.

1. POLYOMINOES

1.1. **Manipulate polyominoes.** You first need code to manipulate polyominoes.

Task 1. *Implement a representation of polyominoes supporting the following functionalities:*

- *creation of a polyomino from a string and creation of a list of polyominoes from a text file;*
- *drawing;*
- *elementary transformations such as translations, rotations (quarter-turn around the origin), reflections (with respect to a coordinate axis), and dilations;*

A polyomino is represented as a set of squares. The text format for a polyomino input is `[(x1,y1), (x2,y2), ..., (xn,yn)]` where `(xi,yi)` are the coordinates of the squares of the polyomino. For the pictures, you can either write your own interface or use the file `Image2d.java`. For example, your implementation should be able to read the file `polyominoesINF421.txt` containing the lines

FIGURE 2. Some particular polyominoes given in the file `polyominoesINF421.txt`.

```
[(0,0), (0,4), (1,0), (1,1), (1,2), (1,3), (1,4), (2,0), (2,4)]
[(0,0), (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (2,1), (2,2), (3,0), (3,1), (3,2), (3,3), (3,4)]
[(0,0), (0,1), (0,2), (0,3), (0,4), (1,2), (1,4), (2,4)]
[(0,1), (0,2), (0,3), (0,4), (1,1), (2,0), (2,1), (2,2)]
[(0,0), (0,3), (0,4), (1,0), (1,1), (1,4), (2,0), (2,1), (2,2), (2,4), (3,0), (3,2), (3,3), (3,4)]
[(0,0), (0,3), (1,0), (1,1), (1,2), (1,3), (1,4), (2,0)]
```

and to print the polyominoes of Figure 2.

1.2. **Generate polyominoes.** In the next part, we will often manipulate all polyominoes of a given area. You therefore need to generate them all. We will consider three types of equivalences on polyominoes:

- *fixed polyominoes* are considered equivalent only if they differ by a translation.
- *free polyominoes* are considered equivalent if they differ by any isometry.
- *one-sided polyominoes* are considered equivalent if they differ by a direct isometry.

Task 2. *Write methods that generate all fixed polyominoes and all free polyominoes. How far can you push these enumerations?*

To enumerate further all fixed polyominoes, we suggest the method of D. H. Redelmeier [Red81] (see in the file `Redelmeier_CountingPolyominoes.pdf`). His method explores a tree of fixed polyominoes without ever testing the equality of two fixed polyominoes. The free polyominoes are then obtained by counting only one canonical representative in each class.

Task 3. *Implement this method to generate further all fixed polyominoes and all free polyominoes. Compare to your solution.*

2. POLYOMINO TILINGS AND THE EXACT COVER PROBLEM

2.1. **The exact cover problem.** Given a ground set X and a collection $\mathcal{C}$ of subsets of X, an *exact cover* is a subcollection $\mathcal{P} \subseteq \mathcal{C}$ which forms a partition of X. For example, for the ground set $X = \{1, 2, \dots, 7\}$ and the collection $\mathcal{C} = \{\{3,5,6\}, \{1,4,7\}, \{2,3,6\}, \{1,4\}, \{2,7\}, \{4,5,7\}\}$, there is a unique exact cover given by the partition $X = \{3,5,6\} \sqcup \{1,4\} \sqcup \{2,7\}$.

We represent an exact cover problem by a 0/1-matrix $M(X, \mathcal{C}) = [m_{S,x}]_{S \in \mathcal{C}, x \in X}$ where for any subset $S \in \mathcal{C}$ and any element $x \in X$, we have $m_{S,x} = 1$ if $x \in S$ and $m_{S,x} = 0$ if $x \notin S$. A solution thus corresponds to a subset of rows which sum to $[1, \dots, 1]$. For example, for the ground set $X = \{1, 2, \dots, 7\}$ and the collection $\mathcal{C} = \{\{3,5,6\}, \{1,4,7\}, \{2,3,6\}, \{1,4\}, \{2,7\}, \{4,5,7\}\}$, the corresponding matrix is

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{1}$$

The exact cover problem is known to be NP-complete. Our goal is to implement fast algorithms for small instances of this problem. The natural method is to use the following backtracking approach:

```
exactCover(X, C) =
  if X = ∅ then return {∅};
  Choose an element x ∈ X to cover first;
  P = {};
  for S ∈ C such that x ∈ S do
      X* = X; C* = C;
      for y ∈ S do
          X* = X* ∖ {y};
          for T ∈ C such that y ∈ T do
              C* = C* ∖ {T};
      for P ∈ exactCover(X*, C*) do
          P = P ∪ {S}; P = P ∪ {P};
```

Task 4. *Implement this algorithm to solve the exact cover problem. Test your algorithm on the exact cover problem* (1) *as well as on large instances (such as all subsets, or all subsets of size k of a ground set with n elements). How far does your algorithm solve the exact cover problem?*

Note that we have some choice for the element $x \in X$ to cover first. A good heuristic to speed up the algorithm is to limit the number of branches at each step, and thus to choose $x \in X$ such that the number of $S \in \mathcal{C}$ with $x \in S$ be minimal. Does it improve the speed of your implementation?

2.2. **D. Knuth's dancing links algorithm.** The problem of a naive implementation of the previous backtracking algorithm is that we keep copying X and $\mathcal{C}$ while trying all possible solutions. D. Knuth gave an elegant solution to this problem[1] using what he called *dancing links* [Kut00].

Assume that you have a doubly linked list where each element x has a pointer $x.L$ to its predecessor and a pointer $x.R$ to its successor. Then you can easily remove x from your list by writing $x.L.R = x.R$ and $x.R.L = x.L$ and conversely you can reinsert x in its former position in the list by writing $x.L.R = x$ and $x.R.L = x$. These simple operations avoid the time-consuming task of copying the list.

The idea of the dancing links algorithm is to use similar update operations on the matrix $M(X, \mathcal{C})$ of an exact cover problem. First, the exact cover problem is transformed into a dancing links data structure as follows:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

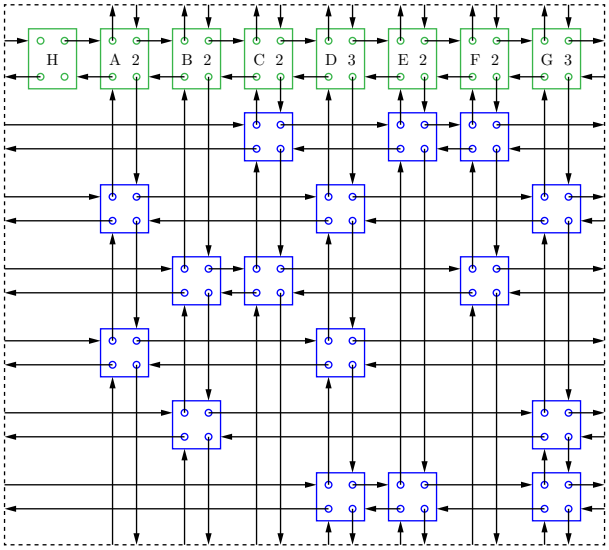

FIGURE 3. The dancing links data structure (the fields C are not represented).

[1] All Polytechnique students should be grateful to D. Knuth for calling this algorithm the "Algorithm X".

Each 1 in the matrix $M(X,\mathcal{C})$ is replaced by a *data object* (blue in Figure 3) with five fields:

- U (up), D (down), L (left), R (right) which point to the four next 1's on top, bottom, left and right of that 1.
- C (column) points to the column object corresponding to the column of this 1.

Moreover, each column gets as header a *column object* (green in Figure 3), which is a special data object with two additional fields:

- S (size) records the number of remaining 1's in the column,
- N (name) records a name for the column.

Task 5. *Implement the dancing links data structure, and write code that transforms an exact cover problem to its corresponding dancing links data structure.*

With this data structure, it is now very efficient to cover or uncover an element using dancing links. Namely, imagine that an element $x \in X$ gets covered by some subset $S \in \mathcal{C}$. We then have to remove all $T \in \mathcal{C}$ containing x. Reciprocally, when backtracking, we will need to reinsert all $T \in \mathcal{C}$ containing x. On the dancing links structure, the element x is a column object, and covering/uncovering x can easily be done by the following algorithms:

coverColumn(x) =
  $x.R.L = x.L$; $x.L.R = x.R$;
  for $t = x.D,\ x.D.D,\ \ldots$ **while** $t \neq x$ **do**
    for $y = t.R,\ t.R.R,\ \ldots$ **while** $y \neq t$ **do**
      $y.D.U = y.U$; $y.U.D = y.D$;
      $y.C.S = y.C.S - 1$;

uncoverColumn(x) =
  $x.R.L = x$; $x.L.R = x$;
  for $t = x.U,\ x.U.U,\ \ldots$ **while** $t \neq x$ **do**
    for $y = t.L,\ t.L.L,\ \ldots$ **while** $y \neq t$ **do**
      $y.D.U = y$; $y.U.D = y$;
      $y.C.S = y.C.S + 1$;

Figure 4 represents the dancing links data structure of Figure 3 after covering B. Note that the uncovering operation undoes what the covering operation did exactly in the reverse order.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

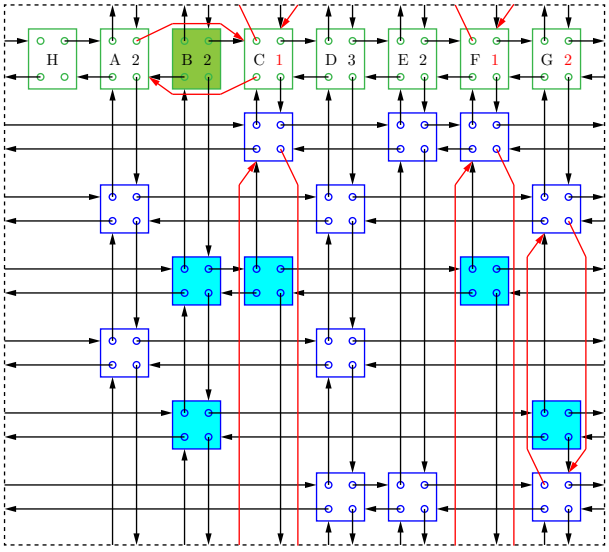

FIGURE 4. Updating the dancing links data structure when covering B.

We can now use the dancing links data structure to obtain an efficient implementation of the backtracking algorithm for exact cover:

exactCover(H) =
  if $H.R = H$ **then return** $\{\varnothing\}$;
  $\mathcal{P} = \{\}$;
  Choose a column x with $x.S$ minimal; **coverColumn**(x);
  for $t = x.U,\ x.U.U,\ \ldots$ **while** $t \neq x$ **do**
    Select t to cover x;
    for $y = t.L,\ t.L.L,\ \ldots$ **while** $y \neq t$ **do**
      coverColumn(y);
    $\mathcal{P} = \mathcal{P} \cup \big\{P \cup \{t\} \mid P \in$ **exactCover**$(H)\big\}$;
    for $y = t.R,\ t.R.R,\ \ldots$ **while** $y \neq t$ **do**
      uncoverColumn(y);
  uncoverColumn(x);
  return $\mathcal{P}$;

Task 6. *Implement the dancing links algorithm to solve the exact cover problem. Test your implementation on the exact cover problem (1) as well as on large instances (such as all subsets, or all subsets of size k of a ground set with n elements). How far does your implementation solve the exact cover problem?*

See [Kut00] for more details on the algorithm if needed (see in the file `Knuth_DancingLinks.pdf`).

2.3. **From polyomino tilings to exact cover.** We now come back to our initial polyomino tiling problem. We consider different instances of this problem:

- we sometimes want to allow to rotate or reflect polyominoes, sometimes not;
- we sometimes want to use exactly once each polyomino, but sometimes allow to not use or reuse a polyomino.

Task 7. *Observe that the problem of tiling a polyomino P using some polyominoes of a set S with possible repetitions can be represented by an exact cover problem where the ground set is the collection of all squares of P, and the subsets correspond to the squares of P covered by a polyomino of S placed at a certain position. Explain how to adapt this representation so that each polyomino of S is used exactly once.*

Task 8. *Implement a method that transforms a polyomino tiling problem (allowing or not rotations, symmetries and reusable tiles) to an exact cover problem. Apply for example to solve the following polyomino tiling questions:*

- *Find all tilings of the polyominoes of Figure 5 by all free pentaminoes.*
- *For a given n (say $n = 4$, 5, or even maybe 6), find all tilings of a rectangle by all fixed (resp. free, resp. one-sided) polyominoes of area n.*
- *For a given n and k, find all polyominoes P of size n which can cover their own dilate kP. How many do you find for $(n, k) = (8, 4)$?*

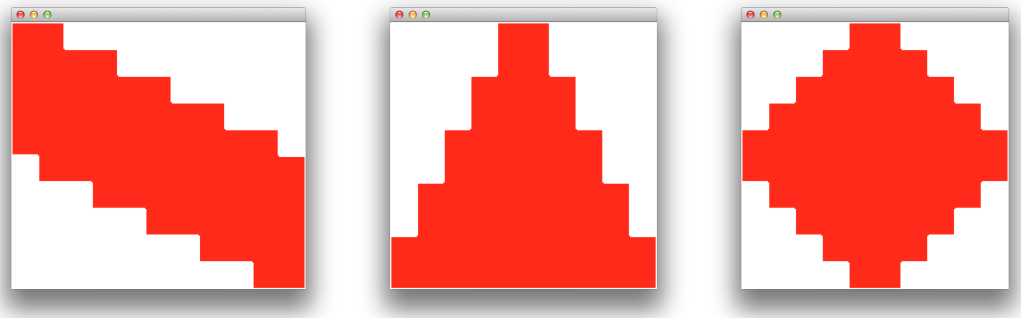

FIGURE 5. Find all tilings of these polyominos by all free pentaminoes.

3. Extensions

The following possible extensions are not mandatory but will be appreciated. Try to choose some of them of your taste. Do not hesitate to propose other extensions than those discussed here.

3.1. **Higher dimension.** A d-dimensional *polycube* is a collection of d-dimensional cubes connected along their codimension 1 faces.

Task 9. *Adapt your code to handle polycube tilings. Find all tilings of the* $2 \times 2 \times 7$ *box by the* 7 *free* 3*-dimensional polycubes of volume* 4*.*

We recall that the group of symmetries of the d-dimensional cube is $\{f_{\pi,S} \mid \pi \in \mathfrak{S}_d, S \subseteq [d]\}$ where $f_{\pi,S}(x_1, \dots, x_d) = \big((-1)^{1\in S} x_{\pi(1)}, \dots, (-1)^{d\in S} x_{\pi(d)}\big)$. If desired, 3-dimensional polycubes can be represented using the library `java3d`, see http://www.java3d.org/.

3.2. **Other lattices.** A different direction is to consider polyominoes on other lattices, in particular on the hexagonal and triangular lattice. These lattices are dual to each other as illustrated in Figure 6.

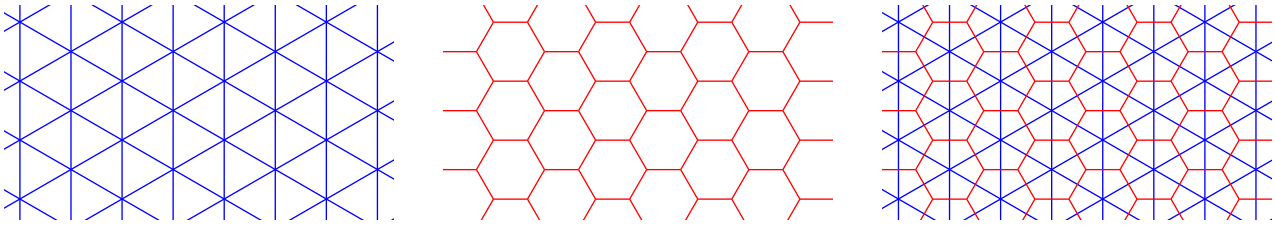

Figure 6. The triangular lattice (left) and the hexagonal lattice (middle) are dual to each other (right).

We consider the following generalizations of polyominoes:

- *hexagonaminoes* are sets of hexagons in the plane that are connected along edges (dually, they are finite connected induced subgraphs of the triangular lattice),
- *triangulaminoes* are sets of triangles in the plane that are connected along edges (dually, they are finite connected induced subgraphs of the hexagonal lattice).

Examples of hexagonaminoes and triangulaminoes are illustrated in Figures 7 and 9, and some hexagonamino and triangulamino tilings are illustrated in Figures 8 and 10.

Task 10. *Adapt your code to deal with polyomino tilings of the triangular lattice and/or of the hexagonal lattice. How many tilings as in Figures 8 and 10 can you find?*

For representation purposes, it might be useful to see the triangular lattice as the 3-dimensional grid projected along the direction $(1, 1, 1)$.

3.3. **Other exact cover problems.** Many other problems can be seen as exact cover problems. The most famous one is certainly the classical sudoku.

Task 11. *Observe that a sudoku grid can be seen as a solution to an exact cover problem. What size do you need for the ground set* X *and the collection* $\mathcal{C}$ *of subsets of* X*? Implement a sudoku solver using your dancing links algorithm.*

There are many other exact cover problems, such as perfect matchings, tripod packing, etc. Choose any of those if you prefer.

References

[Kut00] Donald E. Kuth. Dancing links. Preprint, arXiv:cs/0011047, 2000.
[Red81] D. Hugh Redelmeier. Counting polyominoes: yet another attack. *Discrete Math.*, 36(2):191–203, 1981.

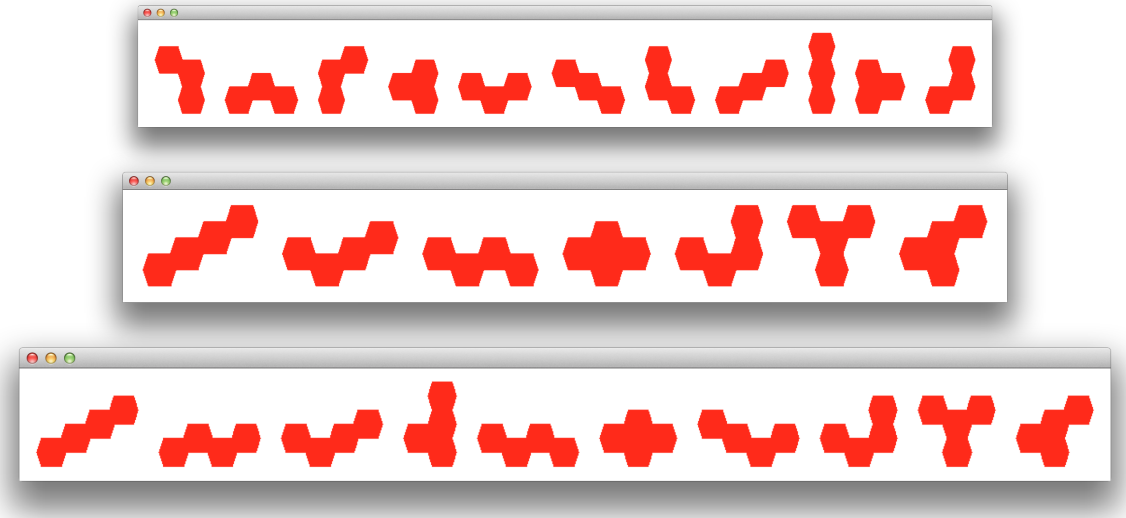

FIGURE 7. Hexagonaminoes: the 11 fixed hexagonaminoes of area 3 (top), the 7 free hexagonaminoes of area 4 (middle), and the 10 one-sided hexagonaminoes of area 4 (bottom).

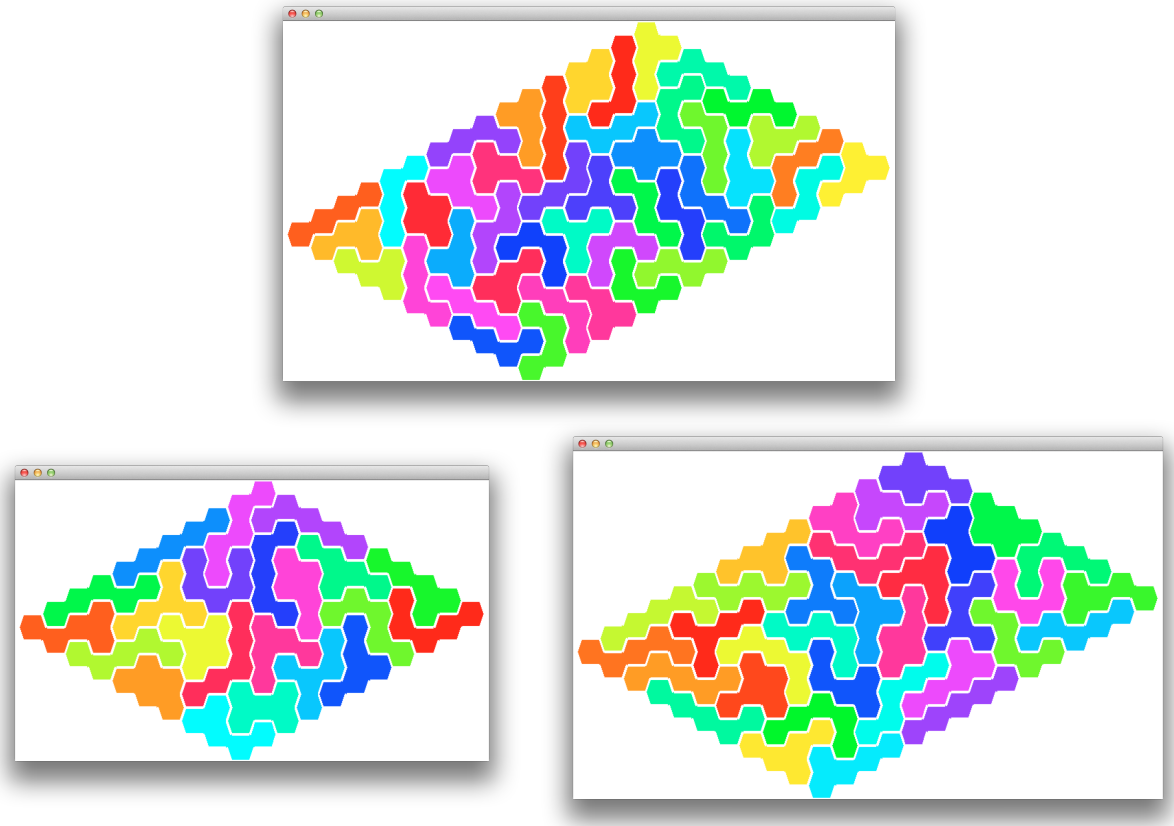

FIGURE 8. Hexagonamino tilings using the 44 fixed hexagonaminos of area 4 (top), the 22 free hexagonaminoes of area 5 (bottom left), and the 33 one-sided hexagonaminoes of area 5 (bottom right). Can you find other such tilings? How many?

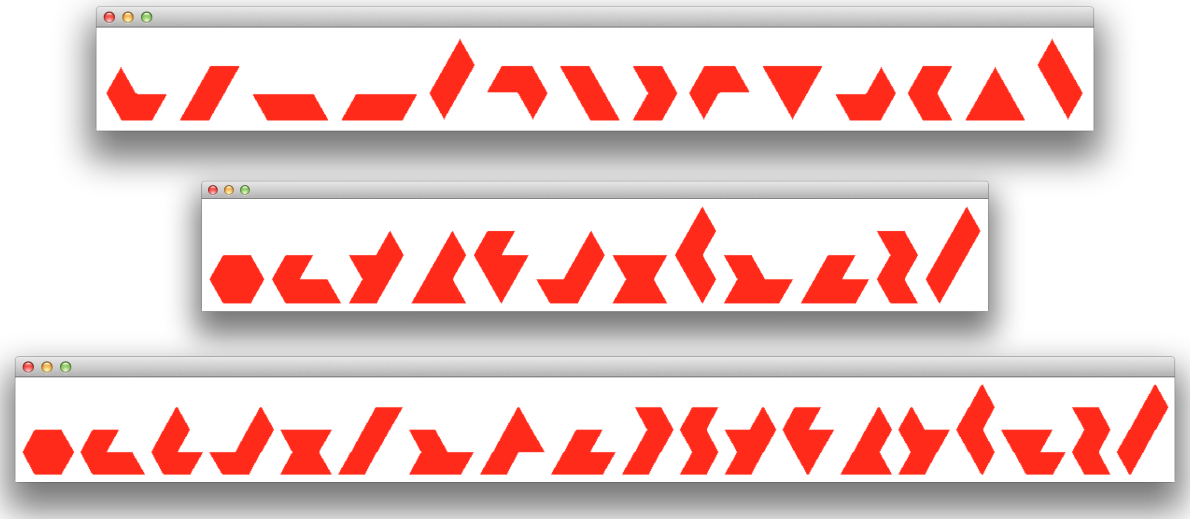

FIGURE 9. Triangulaminoes: the 14 fixed triangulaminoes of area 4 (top), the 12 free triangulaminoes of area 6 (middle), and the 19 one-sided triangulaminoes of area 6 (bottom).

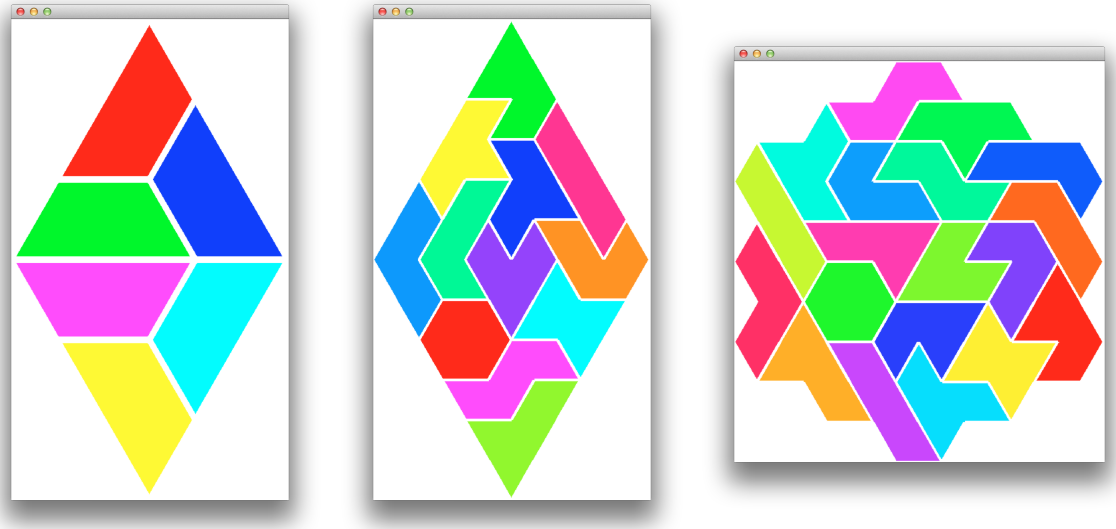

FIGURE 10. Triangulomino tilings using the 6 fixed triangulaminos of area 3 (left), the 12 free triangulaminoes of area 6 (middle), and the 19 one-sided triangulaminoes of area 6 (right). Can you find other such tilings? How many?