Polyomino tiling and exact cover

Thomas Cambier
Thibault Dardinier

January 9, 2017

Contents

Introduction

I Polyominoes

I.1 The Square class

In order to efficiently represent and manipulate polyominoes, which are represented as a set of squares, we needed an efficient representation of those squares. A square has unsurprisingly two main attributes, $(int)x$ and $(int)y$. It also has a third attribute, $(Color)color$, which is only used to efficiently draw polyominoes.

In order to be able to use efficient data structures such as sets for polyominoes, we override *equals* and *hashCode*[1] methods: two squares are equal if and only if their coordinates are the same. We also have generic methods we will discuss when talking about generating symmmetrical and rotationnal polyominoes.

Eventually, we have methods to manipulate squares of polyominoes, such as translations of squares, dilatation of one square (ie. translation), reflections, rotations... These methods don't make sense for a square alone, but they make sense when thinking about a polyomino. We use squares as squares, of course, but also as vectors (of translation for instance), and even for storing data such as dimensions (width and height).

I.2 The Polyomino class

A polyomino is represented by a set of squares, which is his main attribute. As well as for the Square class, we had to override *equals* and *hashCode*: two polyominoes are equal if and only if their squares are the same, the hashCode is the hashCode of the set[2].

There are a lot of methods for manipulating polyominoes, such as translation, dilatation, $\frac{\pi}{2}$ rotation... which are related to the ones previously defined for the squares. We can also create an empty polyomino, or create a polyomino from a *String*. We can check whether a polyomino is included in another one, whether a polyomino is a polyomino (*ie.* its squares are connected).

I.3 Manipulate polyominoes

The Manipulate class contains the useful functions for creating polyominoes from a file, drawing a list of polyominoes side by side (on figure 1 for instance), or drawing the polyominoes with their given coordinates, to see solutions of tiling problems.

It also contains the functions we created to generate fixed and free polyminoes in a given rectangle, Redelmeier's function to generate fixed polyominoes of a given size, and a bunch of functions to generate symmetrical and rotational polyominoes.

[1]The hashcode we implemented is not really efficient, because it returns 1 everytime. We had an issue with another hashcode method we previously implemented, two polyominoes with the same set of squares could be in a set. It is of course possible, with more time, to handle this issue by understanding where in our code we change a square when it is already in a polyomino which is itself in a set.

[2]Since the hashCode of a set if the sum of the hashCodes of his elements, it is in our case the size of the set.

Figure 1: Six random polyominoes with random colors, side to side.

Figure 2: All free polyominoes contained in a square of size 3.

II Generating polyominoes

II.1 A naive generator for fixed and free polyominoes

To generate all the fixed polyminoes in a given area, we simply proceed in a few simple steps. First of all, we generate every possible combination of squares in the given area: for each square in the area, we add it or not to the already existing set of squares. We then have to check for every of these combinations if it is connected[3], and if it is, we put it in canonical form[4], and add it to a set[5]. At the end of this algorithm, we get all the different fixed polyominoes in the given area.

To compute all the free polyominoes with the same constraint (as we can see on figure 2), we first compute the previous function with the same area. Then, we use a set, and for every polyomino we get, we generate its four rotations, and its four rotations after a symmetry. If one of its form is in the set, we do not add it, otherwise we add it.

[3]We use a simple BFS to check if the squares are connected

[4]We simply put it at most up and left.

[5]Thanks to the *equal* and *hashCode* methods we overrode.

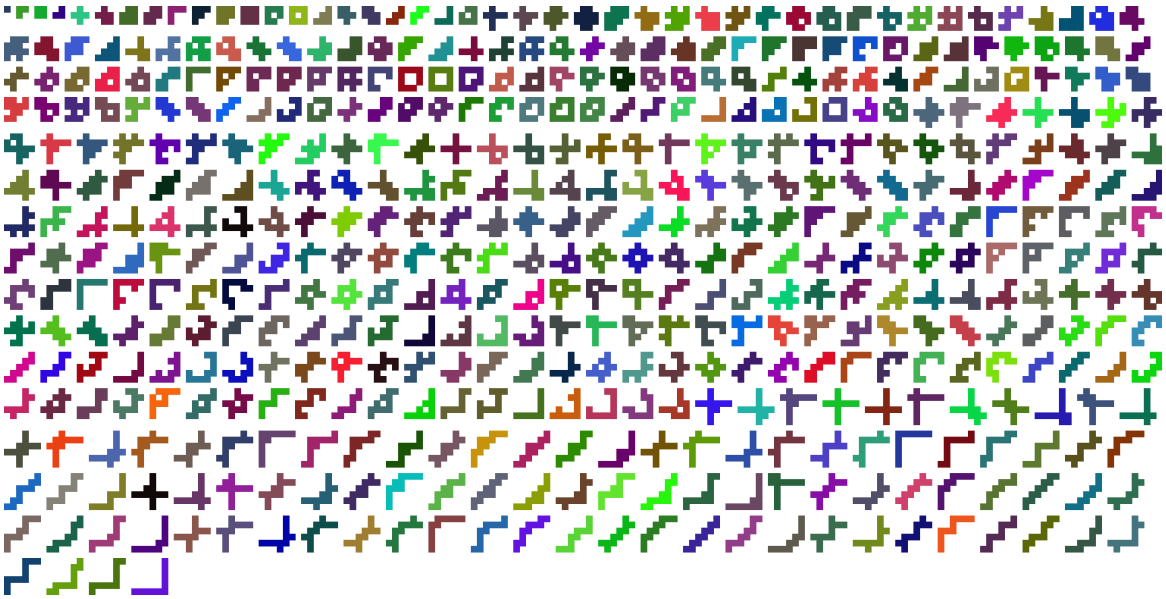

Figure 3: All free polyominoes stable by reflection in descending diagonal, sized under 12.

II.2 Generating fixed polyominoes with Redelmeier's method

II.3 Generating symmetrical and rotational polyominoes

We do not compute all types of symmetrical polyominoes. We compute polyominoes symmetrical with regard to the horizontal axis, the vertical axis, the ascending and descending diagonal, and the ones symmetrical with regard to a point. Symmetrical polyominoes with regard to the vertical axis (and when the axis is between squares), and with regard to the horizontal axis (when the axis is between squares), that is to say HX and VX polyominoes, are computed using fixed polyominoes of half size. All the other symmetrical and rotational polyominoes, and fixed polyominoes are all generated using the same generic function. We only have, for each configuration, to give methods to generate images of the point (none for fixed, three for $\frac{\pi}{2}$ rotations, and one for the others), and a method to know whether a point is allowed or not in the canonical form. These methods are implemented in the Square interface.

III Exact cover and tiling

III.1 Exact cover and the dancing links algorithm

We started to solve the exact cover problem by randomly choosing the first element to cover. Then, we limited the number of reachable possibilities at every step by carefully choosing the first element to cover[6] which sped up the algorithm. At first we were returning all the covers found by the algorithm even though some of them were not exact covers. To improve that, we checked the exactness of every cover returned and finally chose to stop at the first exact cover met for efficiency reasons. The class ExactCover also contains useful functions to manipulate matrices.

[6]We choose the first element such that the number of subsets containing this element is minimal

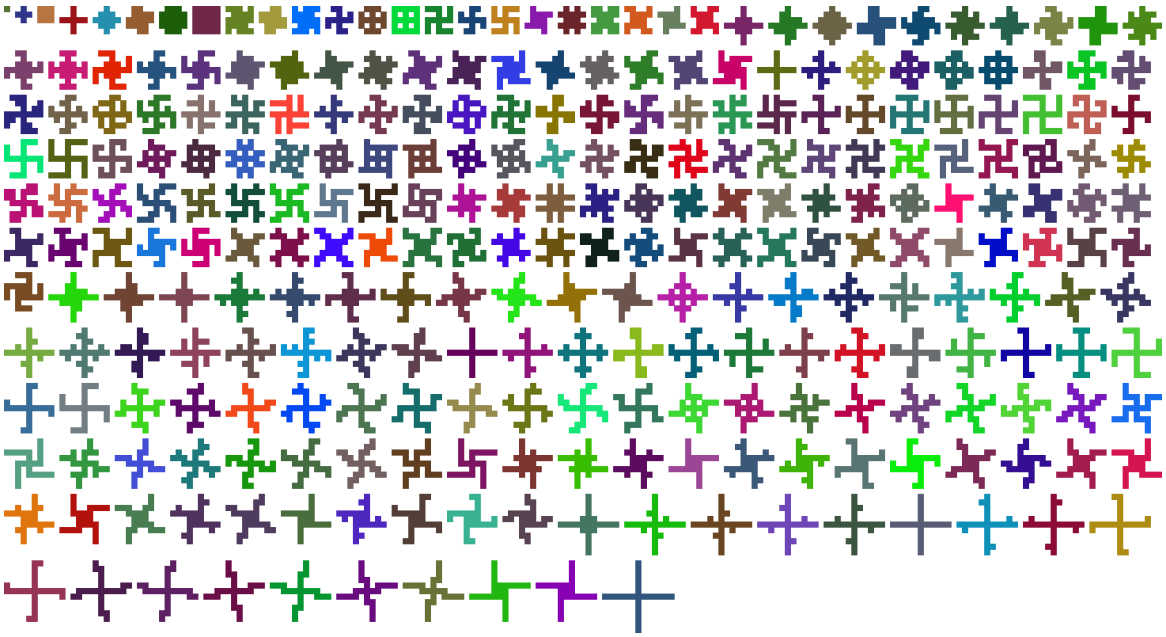

Figure 4: All free polyominoes stable by $\frac{\pi}{2}$ rotation, sized under 25.

However this algorithm was still far too slow for solving exact cover problems, hence we implemented the dancing links structure. This structure allowed us to solve larger instances of the exact cover problem. As well as for the first algorithm, we chose to stop once a solution is found to improve the efficiency.

III.2 From the polyomino tiling problem to exact cover