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 overrode *equals* and *hashCode*[1] methods: two squares are equal if and only if their coordinates are the same. We also have generic methods we'll 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 square, 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 for the Square class, we also have 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].

A polyomino has a lot of methods to be manipulated, 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 contained the useful functions for creating polyominoes from a file, drawing a list of polyominoes side by side (on figure 1 for instance), or draw 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 symmetric and rotational polyominoes.

[1]The hashcode we implemented wasn't really efficient, because it returns everytime 1. 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's already in a polyomino which is itself in a set.

[2]Since the hashCode of a set if the sum of the hashCode of his elements, it's 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 fixed 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's 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 firstly compute the previous function with the same area. Then, we use a set, and for every polyomino we get, we generate his four rotations, and his four rotations after a symetry. If one of his form is in the set, we don't 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.

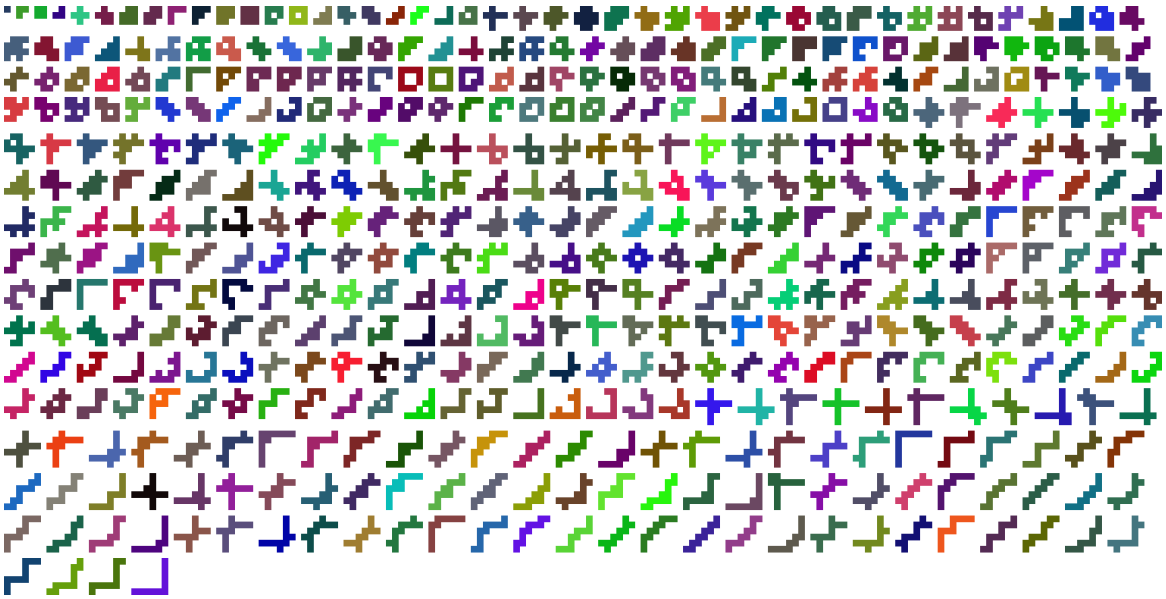

Figure 3: All free polyominoes stable by reflection in descending diagonal, sized under 12.

II.2 Generating fixed polyominoes with Redelmeier's method

II.3 Generating symmetrical and rotational polyominoes

We do not compute all types of symmetrical polyminoes. We compute

III Exact cover and tiling

III.1 Exact cover and the dancing links algorithm

III.2 From the polyomino tiling problem to exact cover

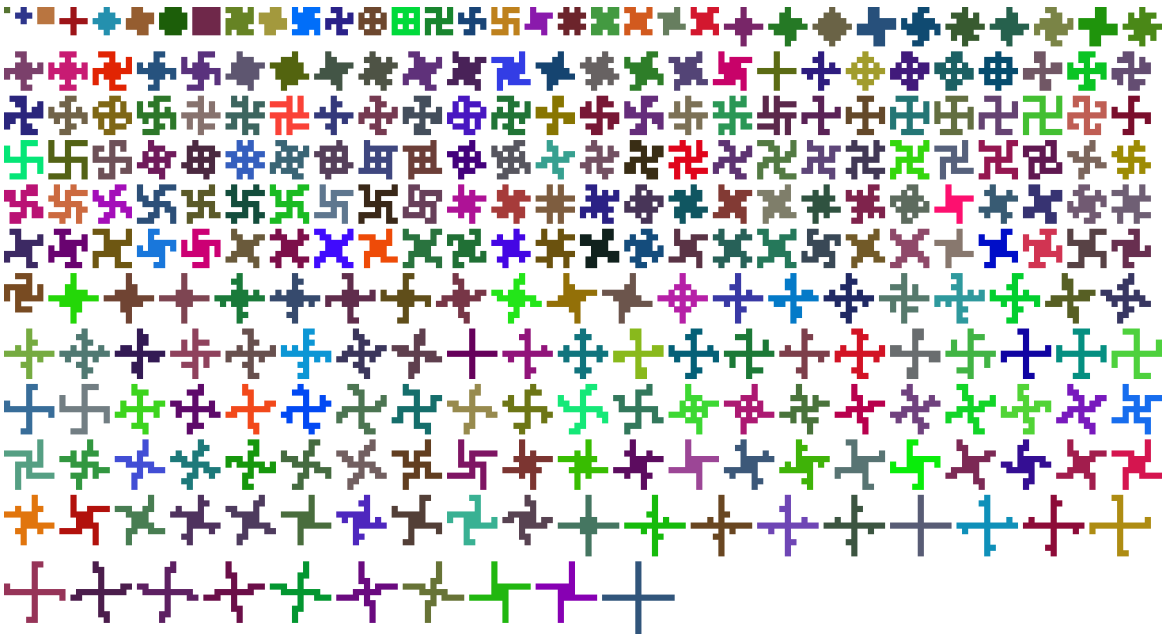

Figure 4: All free polyominoes stable by $\frac{\pi}{2}$ rotation, sized under 25.