INF421 - PROGRAMMING PROJECT

Polyomino tilings & exact cover

8 janvier 2017

Guillaume Dalle
Clément Mantoux

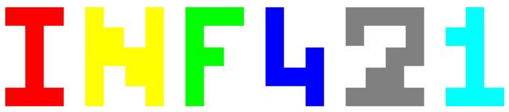


Figure 1 – Creating polyominoes from a file

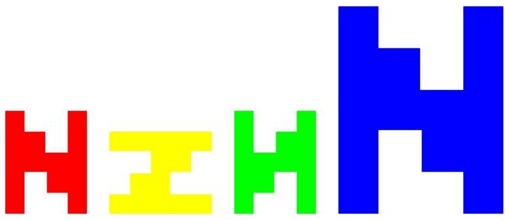


Figure 2 – Rotation, symmetry, dilatation

1 Polyminoes

Task 1

We chose to represent polyominoes as a matrix of booleans, with cases containing true or false depending on whether the respective squares are part of the polyomino or not. The convention is that these polyominos should "touch" every border of the matrix, so that no space is wasted. Furthermore, this means that there is only one possible representant of each translation class, so we can't generate a fixed polyomino twice.

We spent some time hesitating between this format and a LinkedList containing only the coordinates of the polyomino's cases, but it turns out a matrix is easier in 2 dimensions and for square lattices, while a list might enable us to easily generalize to more complicated settings.

The functions enabling translations, rotations, symmetries and dilations are pretty straightforward with this representation. Here's an example of what we obtained :

Task 2

To enumerate all polyominoes of size n we chose a very naive method :

1. Enumerate all polyominoes of size $n-1$
2. For each of those, create a new polyomino of size n by adding a square at a valid position
3. Check if this new polyomino has already been obtained before, and if not add it to the list

The nuance between free and fixed polyominoes is found in step 3, that's where we use different criteria to say whether a given polyomino is already in the list (up to any isometry or just translations). Here's what we obtain when looking for free polyominoes :

n	Free polyominoes	Naive method
4	5	4 ms
5	12	3 ms
6	35	15 ms
7	108	87 ms
8	369	764 ms
9	1285	9973 ms

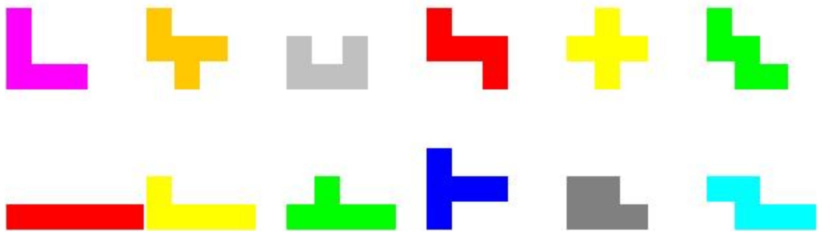


Figure 3 – Free polyominoes of size 5

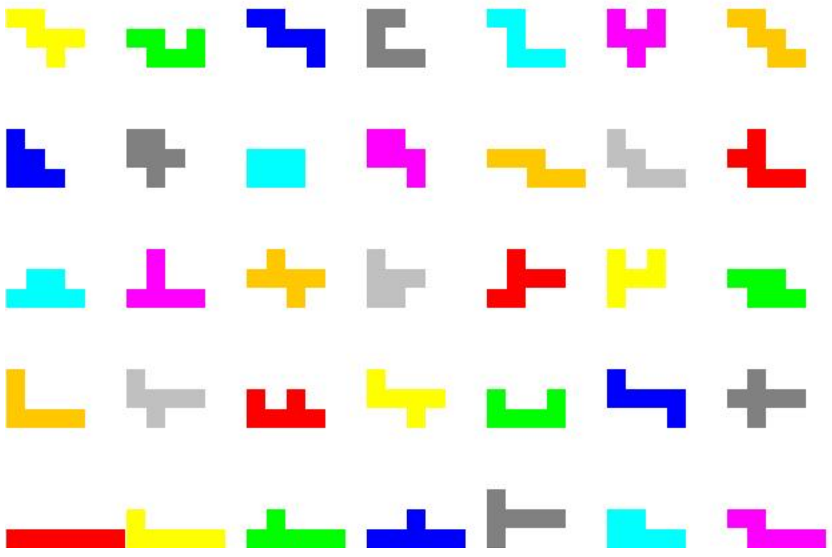


Figure 4 – Free polyominoes of size 6

Task 3

Indeed, Redelmeiers method is a lot faster for the generation of both fixed and free polyominoes. In order to implement the Redelmeier version of the free polyomino counting algorithm, we saved polyominoes in a HashMap using tiles-based hash keys. Here is the comparison of both algorithms :

n	Fixed	Free	Naive Fixed	Naive Free	Redelmeier Fixed	Redelmeier Free
4	19	5	1 ms	4 ms	0 ms	5 ms
5	63	12	3 ms	3 ms	1 ms	4 ms
6	216	35	7 ms	14 ms	2 ms	7 ms
7	760	108	15 ms	80 ms	5 ms	15 ms
8	2725	309	195 ms	805 ms	13 ms	36 ms
9	9910	1285	2969 ms	11188 ms	20 ms	217 ms
10	36446	4655	?	?	133 ms	751 ms
11	171714	17073	?	?	742 ms	2066 ms
12	677575	63600	?	?	3512 ms	8196 ms

2 Polymino tilings & the exact cover problem

Task 5

To convert a matrix into a DancingLinks object, we decided to create a matrix of data objects that are all connected to each of their neighbors, and then remove the unnecessary data to get the right result.

Tasks 4 & 6

Here's a comparison of both algorithms on the task of finding the partitions of $[\![1, n]\!]$ with all possible subsets available. We couldn't push the enumeration much further.

n	Partitions of $[\![1, n]\!]$	Naive ExactCover	DancingLinks
3	5	1 ms	3 ms
4	15	1 ms	1 ms
5	52	3 ms	2 ms
6	203	9 ms	4 ms
7	877	10 ms	7 ms
8	4140	52 ms	25 ms
9	21147	268 ms	63 ms
10	115975	1569 ms	701 ms
11	678570	11628 ms	6345 ms

Task 7

The conversion between polyomino tiling and ExactCover is done by creating a matrix where each line is a binary array corresponding to the squares of the ground set covered by one particular polyomino at one particular position in the region of the plane we consider.

To use each polyomino exactly once, we only need to add n columns to the ExactCover matrix, one for each type of polyomino we wish to use. The line corresponding to a polyomino will be the same as before, with an additional one in his column, so that each type is used exactly one time.

Task 8

Tiling of a $s \times s$ square by fixed polyominoes of size s

Here's a first example to demonstrate our methods work : tiling a $s \times s$ square with fixed polyominoes of order s.

s	Number of such tilings
3	10
4	117
5	4006
6	451206

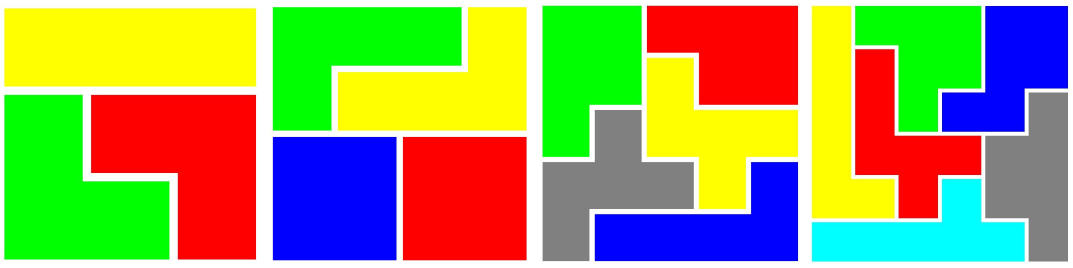


Figure 5 – Tiling of a $s \times s$ square with s-polyominoes

Tiling of a triangle by all free polyominoes of size 5

As we said, it is possible to tile a region using each free polyomino exactly once. If we consider the figures shown in the subject, there are :

- 404 ways to pave the diagonal rectangle
- 374 ways to pave the triangle
- 0 ways to pave the diamond

Covering one's own dilate

There are 369 free polyominoes of size 8. Out of all these, only can cover their own dilatation by a factor of 4 : they are the ones shown on the figure.

Task 10 : Hexagonaminoes counting and exact cover

Implementing Hexagonamino structure

Hexagonamino are implemented as Polyomino : two of the three axis are chosen as X and Y-axis. The one this that changes is the neighborhood of each tile, which has 6 neighbors instead of 4. All polyominoes counting algorithms can be adapted very naturally into hexagonamino counting. Similarly, the exact cover problem can be solved using the polyominoes' method.

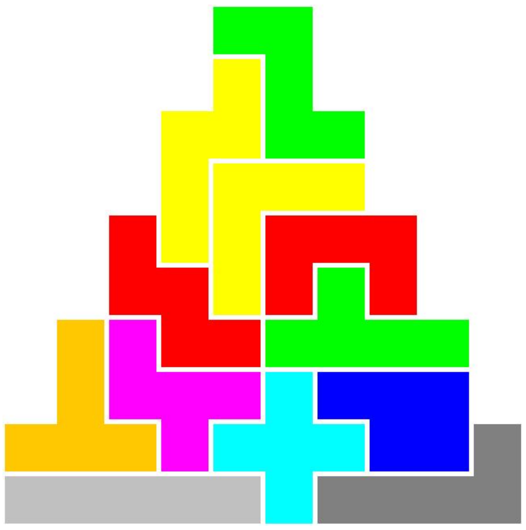


Figure 6 – Tiling of a triangle with all free pentominoes

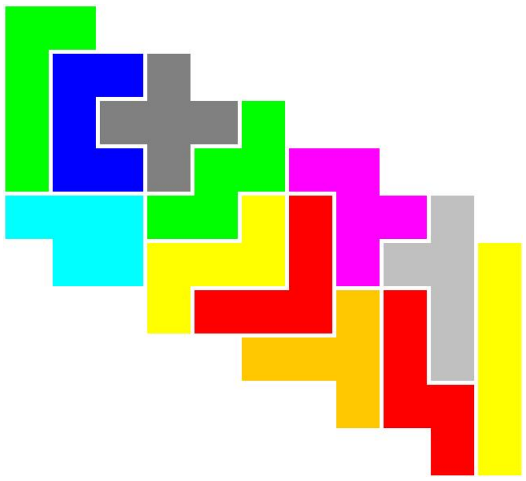


Figure 7 – Tiling of a funny rectangle with all free pentominoes

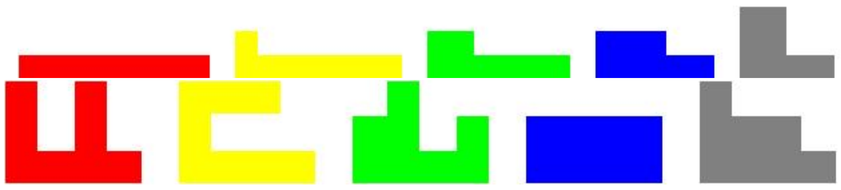


Figure 8 – The 10 octominoes that cover their 4-dilate

Couting Hexagonaminoes

3 Sudoku solving

It is well known that the problem of solving a sudoku boils down to an ExactCover problem with a very specific matrix of size $(729, 324)$:

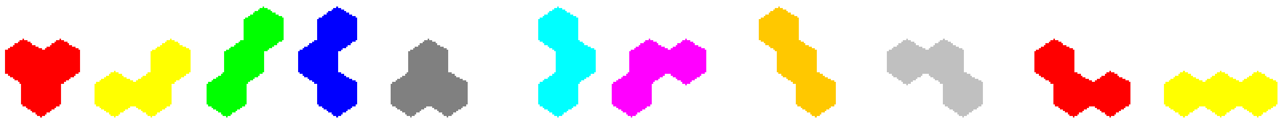


Figure 9 – The 11 fixed hexagonaminoes of area 3

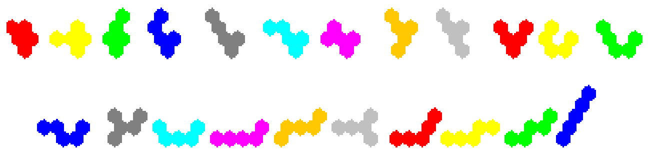


Figure 10 – The 22 fixed hexagonaminoes of area 5

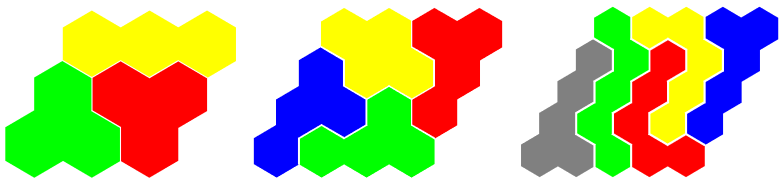


Figure 11 – Tiling of a $s \times s$ square with s-hexagonaminoes

— The columns of that matrix represent four categories of constraints, just like the squares we must fill with polyominoes, but a little more complex (see below)

— The rows of that matrix represent possibilities, that is "one number in one row and one column", they contain almost only zeros, and only 4 ones, because each possibility satisfies 4 constraints, one from each type

And here are the four constraints :

RC : One number only for each couple Row-Column (that is each square of the grid)

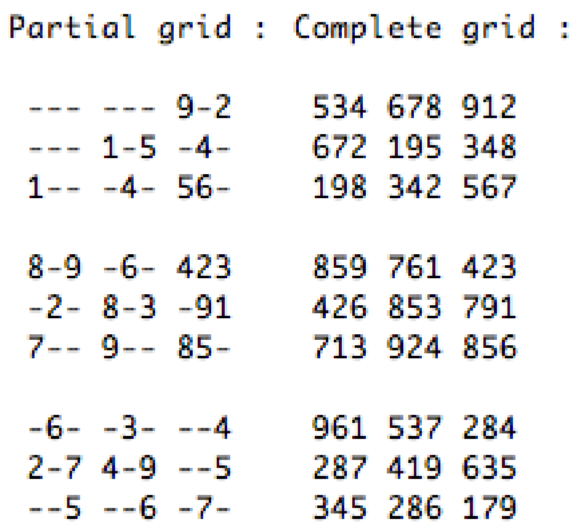
RN : Each Number can appear only one time in a given Row

CN : Each Number can appear only one time in a given Column

BN : Each Number can appear only one time in a given Box, namely one of the nine bigger squares that compose the grid

Which means we have $4 \times 81 = 324$ constraints and $9^3 = 729$ possibilities.

If we run ExactCover on this whole matrix, we will get every possible filled out sudoku grid... after the universe has died. Twice. So to speed things up, we start from a partial grid, which allows us to take into account only the inactive constraints and the still feasible possibilities. As a result, the matrix is much smaller and the algorithm runs in a very reasonable time. As a bonus, we even know whether or not the solution exists and is unique, because the number of partitions can be either zero, one or many.



```
Partial grid : Complete grid :

 --- --- 9-2    534 678 912
 --- 1-5 -4-    672 195 348
 1-- -4- 56-    198 342 567

 8-9 -6- 423    859 761 423
 -2- 8-3 -91    426 853 791
 7-- 9-- 85-    713 924 856

 -6- -3- --4    961 537 284
 2-7 4-9 --5    287 419 635
 --5 --6 -7-    345 286 179
```

Figure 12 – Solving a sudoku