INF 421 PROGRAMMING PROJECT (PI)

Polyomino Tilings and Exact Cover

03 February 2020

Duc Thinh NGO,Chin Wei CHONG

ÉCOLE POLYTECHNIQUE

IP PARIS

1 POLYOMINOES

1.1 Task 1

We represent the object *Polyomino* by two *ArrayLists* with the Integers that indicate the coordinates of the polyomino respectively. Such construction has mainly two advantages, this representation saves the spaces occupied, as compared to a representation by an Array which contains the unnecessary coordinates for the polyomino. Besides, *ArrayList* is an ordered list, which allow us to print and track the points connected one after another. We define a series of methods which allow us to perform the elementary transformation as illustrated below :

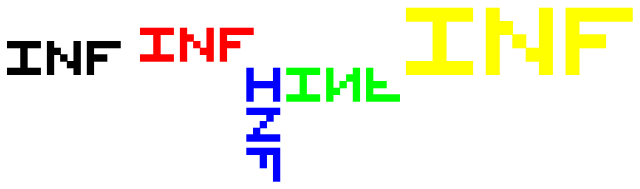


Figure 1 – The properties of the polyomino : translation, rotation, reflection and dilation

For convenience, we also define another class, namely *ListOfPolyonominoes* which allow us to draw a series of *Polyominoes* from a text file, as illustrated below :

Figure 2 – The text file polyominoesINF421.txt

1.2 Task 2

In the class *NaiveGenerator*, we define a method to enumerate the fixed polyomino using a naïve recursive algorithm :

Base Case : *LinkedList* with *Polyomino* of size 1

1. Generate a *LinkedList* which contains all the *Polyomino* of size $n-1$
2. For each *Polyomino* in the *LinkedList*, create *Polyomino* of size n by adding a valid coordinate to the current *Polyomino*
3. Add the newly created *Polyomino* to the resulting *LinkedList* if it is different from the previously created *Polyomino*

This naïve method terminates well as the integer n is reduced in each step until it reaches 1. It is correct too since the recursion only adds square step by step next to the original polyomino (invariant : all squares are connected) up to n (thus the size of the polyomino is correct) and the polyominoes generated are unique (if not they will be rejected in step 3).

To enumerate the free polyomino, we simply make use of the result of the fixed polyomino and filter them according to their symmetries. Do note that the class *Point* is constructed to represent a coordinate conveniently.

The discussion and the summary table of the time spent will be discussed under Task 3.

1.3 Task 3

We follow the algorithm of Redelmeier listed on the research paper and construct the class *Enumeration* accordingly. Different from the previous task, we construct the class *StackOf-Point2D* to represent a stack of coordinates easily. The free polyominoes are generated by filtering the result of fixed polyonomies but they are counted using the information discussed in the research paper. The results of task 2 and task 3 are listed in the table below :

Time required (ms)				
Size	Naïve fixed	Naïve free	Redelmeier fixed	Redelmeier free
4	39	49	2	24
5	120	138	3	45
6	280	415	5	97
7	1859	2613	7	185
8	24250	31400	17	379
9	358517	473501	26	849
10	Unknown	Unknown	55	1760
11	Unknown	Unknown	147	5278

Clearly, with the algorithm of Redelmeier, we can compute the number of fixed and free polyominoes faster. The examples of the results generated are as follows :

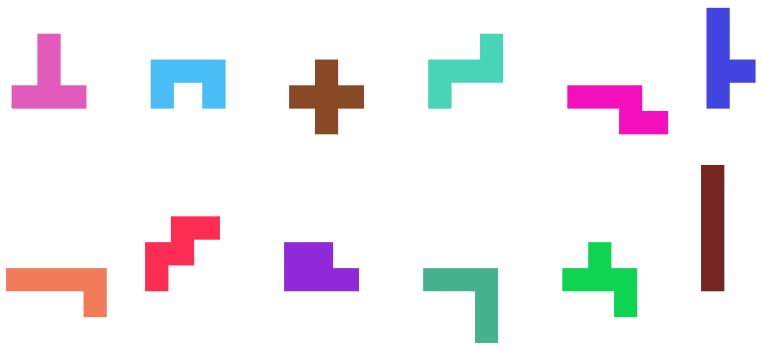


Figure 3 – The free polyominoes of size 5

2 POLYOMINO TILINGS AND THE EXACT COVER PROBLEM

2.1 Task 4

The method solve of the class *ExactCover* is constructed by using the backtracking algorithm illustrated in the handout. We use the data structure *Set* here since there is no order for the choice of ground set $\mathcal{X}$ and the collection $\mathcal{C}$. By wisely fixing the order of choosing the element of the ground set, the speed of implementation is slightly increased. Refer to the results in the table of Task 5 & Task 6.

2.2 Task 5 & Task 6

In order to adapt Knuth's dancing links algorithm, we construct the class *Node*. It gives a representation to the data object and column object as well as associate them with methods such as connecting the object horizontally and removing/recovering the object in the middle of the list.

The constructor of the class *DancingLinks* takes two arguments : ground set $\mathcal{X}$ and collection $\mathcal{C}$, and the method *exactCover* follow exactly the algorithm presented in the handout. Both Task 4 and Task 6 are tested on the problem given in the handout and problems with all subsets of size k of a ground set with n elements (the problem is generated in the *DancingLinks* class). The results are listed below :

Time required (ms)					
n	k	Solutions	*ExactCover* solve	*ExactCover* fastersolve	*DancingLinks*
6	1	1	5	1	0
6	2	15	10	2	1
6	3	10	18	3	1
6	6	1	2	0	0
12	1	1	6	2	0
12	2	10395	19205	17502	84
12	3	15400	39810	37088	63
12	4	5775	10671	6804	37
12	6	462	4727	4201	44
12	12	1	2	0	0
Example given					
-	-	1	2	1	0

Since the algorithm is still the same, the time difference only arises from the different data structure used. The big difference is that Task 4 requires to copy the ground set $\mathcal{X}$ and the collection $\mathcal{C}$, whereas Task 5 only manipulate the nodes by covering and uncovering them.

A little analysis of the algorithm here : The algorithm ends well because each loop reduces the elements of $\mathcal{X}$ and it terminates when $\mathcal{X}$ is empty. The algorithm is also correct : there will be no overlap of elements in $\mathcal{X}$, all used elements will be removed (no extras) ; all elements in $\mathcal{X}$ are considered in the solution (no missing) ; thus we have exactly covered the matrix.

2.3 Task 7

We tackle the problem of tiling by using the representation of *Point*. The ground set $\mathcal{X}$ is formed by the *Points*, each of them indicates the position to be covered by the *Polyominoes*. On the other hand, we use *Points* to represent every polyominoes that will be used to cover. These polyominoes can be generated easily using the *Enumeration* class (according to what we require, fixed/free), and then be translated to generate every single possibility. If we restrict ourselves in the case where the collection $\mathcal{C}$ do not allow repetition, we construct the collection $\mathcal{C}$ by adding an indication factor to each element of collection. The indication factor is chosen to be the original polyomino translated to origin. Note that this indication factor is not included in $\mathcal{X}$ since we do not wish to restrict ourselves in the case that we have to use all polyominoes (Refer to the image below). Here is the tricky part : the *DancingLinks* structure is slightly modified such that all these indication factors are linked, and thus only one polyomino will be chosen in each group. At the end of the execution, we transform the set of *Points* back to the original *Polyomino* and we obtain the desired result as expected.

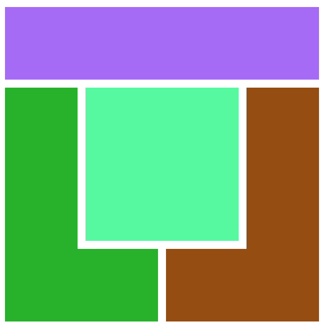


Figure 4 – Not all fixed polyominoes of size 4 are used to construct a square

2.4 Task 8

Following the explanation given in Task 7, we construct the methods in the class *ExactCover*. Using all free pentaminoes, we found that there are 404 ways to cover the tilted rectangle, 374 ways to cover the triangle, and there is no way to cover the diamond. The example of the results are as follows :

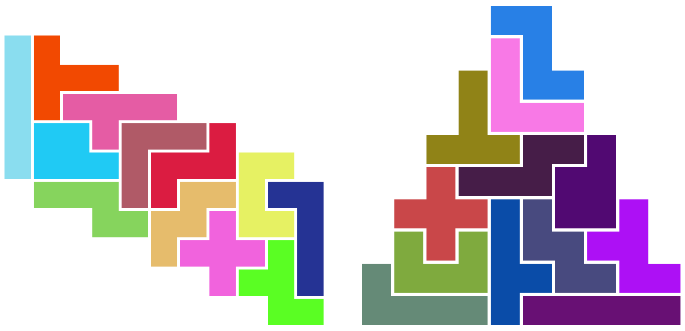


Figure 5 – The tilted rectangle, triangle

To use all the polyominoes of area n to cover all tilings of a rectangle, we find firstly the area of the rectangle that will be formed, which is simply the product of the size of a polyomino and the number of polyominoes with that size. Then we find all the possible rectangles whose sides are integers. By using the same concept on the problem, we found the following results and examples are given below :

Number of solutions			
n	Type	Size	Solutions
4	Free	-	0
4	Fixed	-	0
5	Free	(20,3)	8
5	Free	(15,4)	1472
5	Free	(12,5)	4040
5	Free	(10,6)	9356
5	Fixed	?	?

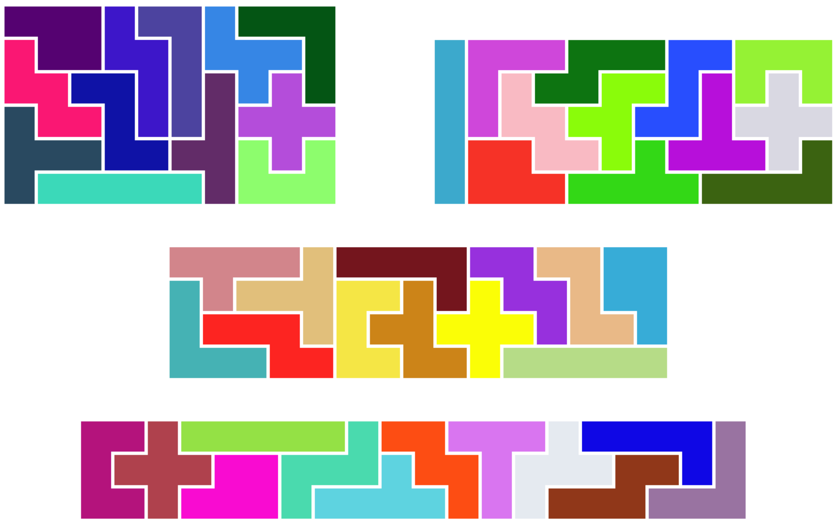


Figure 6 – The rectangles with sides (10,6), (12,5), (15,4), (20,3) covered by all free polyominoes of size 5

For the tilings of own dilate, we define a particular method *tilingOfDilate* in *ExactCover* class and given (n,k) = (8,4), we found 10 results :

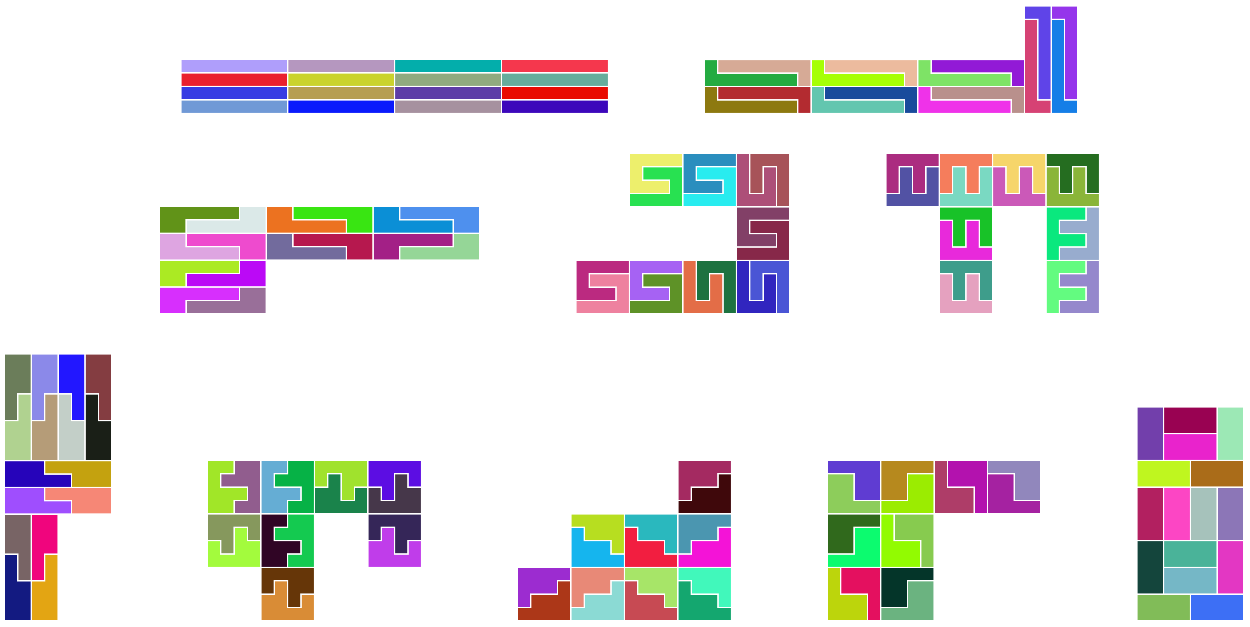


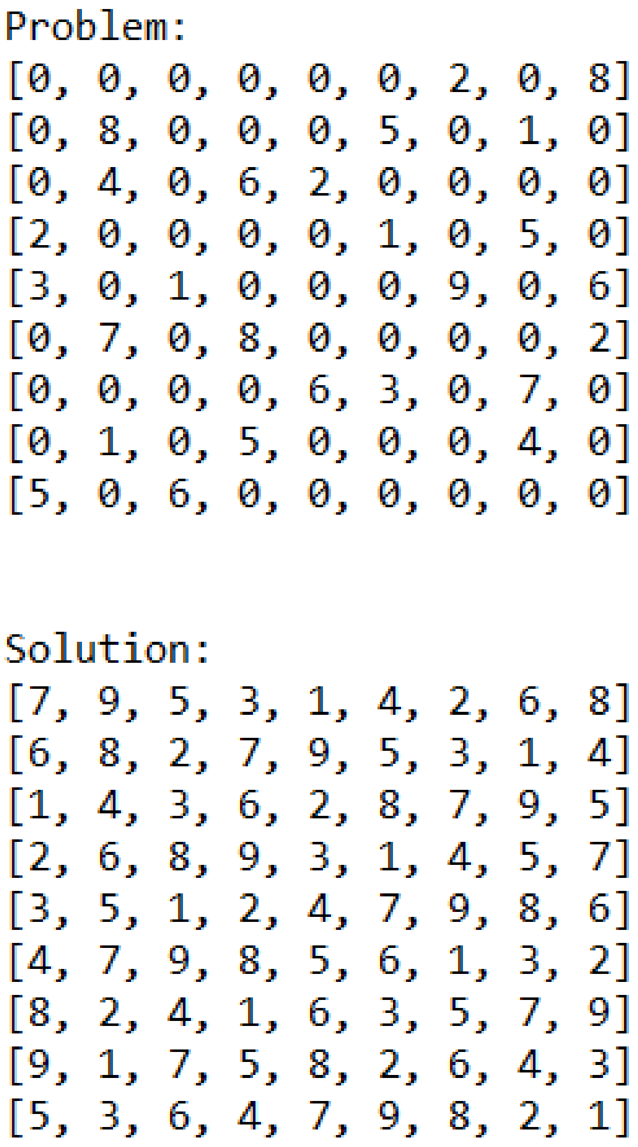
Figure 7 – The results of the tilings of own dilate

3 EXTENSIONS

3.1 Task 11

In the class *Sudoku*, we denote the column, row and the box as $\mathcal{C}$, $\mathcal{R}$ and $\mathcal{B}$. Also, the index of the boxes are defined as ascending order (1-9) from top to bottom for $\mathcal{C}$, from left to right for $\mathcal{R}$, from left to right and then top to down for $\mathcal{B}$. Since each of them have to be covered by one of the number 1-9, we use Axy (A = $\mathcal{C}$,$\mathcal{R}$,$\mathcal{B}$, x,y = 1-9) to indicate that the number y exists in the x^{th} A. Besides, one of the essential constraint of Sudoku is to ensure that one cell accepts exactly a number. In this case we use the representation XY to indicate the position of the cells where X represents the row number and Y represents the column number. Therefore, the ground set $\mathcal{X}$ will be in the form of {$\mathcal{C}$uv,$\mathcal{R}$wx,$\mathcal{B}$yz,XY}. Since u,v,w,x,y,z,X,Y take values from 1-9, we have maximumly $9^2 \times 4 = 324$ elements in the ground set (This number will be reduced with a given grid). Each time we place a number n in c^{th} $\mathcal{C}$, r^{th} $\mathcal{R}$, b^{th} $\mathcal{B}$, we make it as a set {$\mathcal{C}$cn,$\mathcal{R}$rn,$\mathcal{B}$bn,rc}, thus the collection $\mathcal{C}$ will have the form of {{$\mathcal{C}$cn, $\mathcal{R}$rn,$\mathcal{B}$bn,rc}} which has the maximum size of $9 \times 9 \times 9 \times 9 = 729$ (4 freedoms, c,r,b,n = 1-9). By applying the exact cover method, we obtained the desired results :

Time required (ms)	
Normal Exact Cover	Dancing Links
66548	19



```
Problem:
[0, 0, 0, 0, 0, 0, 2, 0, 8]
[0, 8, 0, 0, 0, 5, 0, 1, 0]
[0, 4, 0, 6, 2, 0, 0, 0, 0]
[2, 0, 0, 0, 0, 1, 0, 5, 0]
[3, 0, 1, 0, 0, 0, 9, 0, 6]
[0, 7, 0, 8, 0, 0, 0, 0, 2]
[0, 0, 0, 0, 6, 3, 0, 7, 0]
[0, 1, 0, 5, 0, 0, 0, 4, 0]
[5, 0, 6, 0, 0, 0, 0, 0, 0]


Solution:
[7, 9, 5, 3, 1, 4, 2, 6, 8]
[6, 8, 2, 7, 9, 5, 3, 1, 4]
[1, 4, 3, 6, 2, 8, 7, 9, 5]
[2, 6, 8, 9, 3, 1, 4, 5, 7]
[3, 5, 1, 2, 4, 7, 9, 8, 6]
[4, 7, 9, 8, 5, 6, 1, 3, 2]
[8, 2, 4, 1, 6, 3, 5, 7, 9]
[9, 1, 7, 5, 8, 2, 6, 4, 3]
[5, 3, 6, 4, 7, 9, 8, 2, 1]
```

Figure 8 – The problem and the solution of a solution grid