POLYOMINO TILING AND EXACT COVER

Houssam El Cheairi and Abdelhafid Souilmi

Ecole Polytechnique - I.P Paris

Abstract. The exact cover problem is a classical combinatorics topic with various applications. In this project we study two of these applications : Polyomino tiling and Sudoku resolution. We implement various algorithms and data structures (Dancing Links) to test and optimize the complexity of our programs.

Keywords: Polyominos · Dancing links · Exact cover.

Task 1

In order to represent Polyominos, two classes have been implemented:

1. **Point** : Creates the object "Point" representing a point in the cartesian plane with two integer attributes .x and .y coding its projections over the x-axis and the y-axis
2. **Polyomino**: A polyomino is understood as a set of unit squares in the plane satisfying a central condition :
   None of the squares is isolated or shares a vertex-connection with another square, i.e the squares are "linked" by their edges like dominos.
   The class **Polyomino** codes a Polyomino with two attributes :
   (a) **Squares** : It's a list of **Point** objects, each represents the coding of a single square forming the polyomino. We code a square in the plane by its lower-left Vertex.
   (b) **Color** : Is a **Color** object and codes the color of the polyomino

Then the we define methods to apply different geometric transformations to the polyomino objects. These methods are first defined on the **Point** class and then on the **Polyomino** class. Each of these geometric transformations have a complexity of $\mathcal{O}(n)$ (where n is the size of the polyomino).

We also define methods to convert **String** objects to **Polyomino** object with a complexity of $\mathcal{O}(n)$.

Task 2

Before getting into the details of the generators used, we need to find a way to compare two polyominos. To do so we associate a *canonical* representation to each polyomino, namely its unique representation where the lowest square of

the left side of the polyomino is at the origin of the plane. We can thus check in $\mathcal{O}(n)$ if two polyominos of size n correspond to the same *fixed* or *free* polyomino.

We use the following recursive algorithm to generate fixed polyominos:

Assume we have a list L_n with all the fixed polyominos of size n .In order to generate the polyominos of size $n+1$ we look at each polyomino *polyo* in L_n and create all possible polyominos by adding a new square to *polyo*, these new polyominoes are added to the list L_{n+1} .Finally we check L_{n+1} and delete possible repetitions of polyominoes. Note that each time we add a square, we need to create a copy of the original n-size polyomino, therefore the complexity of adding a single square is $\mathcal{O}(n)$,the complexity of creating all possible new polyominos out of a single polyomino is $\mathcal{O}(n^2)$, and thus the complexity of generating L_{n+1} with repetitions is $\mathcal{O}(n^2 L_n) + C_n$ where C_n is the complexity of generating L_n.

We use a naive approach to check for repetitions, namely a double loop that updates a list with polyominos that we haven't encountered before (in the "fixed" sense or "free" sense). Using some basic math we reach a time complexity of $\mathcal{O}(n^2 L_n + (nL_n)^2) + C_n$, the term $(nL_n)^2$ refers to the complexity of the double loop to check for repetitions. Moreover, Redelmeier states that the size of L_n is exponential, henceforth the total time complexity is exponential as well.

The complexity analysis shows us that the most costly operation is the check for repetitions step, we can optimize this by implementing a hash table that would associate to each two fixed polyominos the same hash value and then use the **HashMap** structure to store the generated polyominos.

Here is a table of the performance of our programs using the following hardware :

- CPU : Intel(R) Core(TM) i7-8750H 2.20 GHz, 2.21 Ghz hexa-core
- RAM : 16,0 Go (We allowed Eclipse to use more Ram resources than usual)
- GPU : Nvidia GeForce GTX 1050 Ti Max-Q

n	Fixed	Exection time (s)	Free	Execution time (s)
1	1	0.043	1	0.029
2	2	0.044	1	0.030
3	6	0.044	2	0.031
4	19	0.045	5	0.034
5	63	0.055	12	0.040
6	216	0.080	35	0.056
7	760	0.44	108	0.180
8	2725	6.7	369	1.5
9	9910	130	1285	22.5

Task 3

Here we implemented the Redemeiler algorithm to generate fixed and free polyominos of a given size. The huge performance difference in comparison to the

naive approach is due to the fact that Redemeiler's algorithm avoids the "check for repetitons" step, which is the most costly step in terms of complexity.

As stated in Redelmeier's article, we need to implement a canonical way to identify a polyomino and one of its isometries. To do so we first choose to identify each polyomino with its "fixed" canonical representation (see Task 1), then we identify this polyomino with a list of pairs of integers (representing the (x,y) coordinates of the unit squares forming the polyomino), however the issue would be that this list depends on the order in which we initially constructed the polyomino. To avoid this dilemma we sort these lists in lexicographical, more precisely we first order the list with respect to the **y** coordinate and then with respect to the **x** coordinate.

Next we generate the **Orbit** of each polyomino i.e all of its images through isometries, compare them (its possible since the lexicographical order we defined is a total order) and choose the smallest (in the lexicographical sense) of these lists as a canonical identifier of the isometries of our initial polyomino.

In practice we implement the "list of pairs" as a list containing lists of size 2, where the first entry is the **x** coordinate, and the second one the **y** coordinate.

The generation of n-size free polyominos is done according to the following steps:

- **Step 1** : Generate the fixed polyominos of size n.
- **Step 2** : Create a HashMap using the hashing procedure described above with lists of integers.
- **Step 3** : For each fixed polyomino compute its canonical representative and add it to the HashMap.
- **Step 4** : Retrieve the values stored in the HashMap object.

Using the same hardware here is a table summarizing the performances of Redemeiler algorithm:

n	Fixed	Exection time (s)	Free	Execution time (s)
5	63	0.045	12	0.059
6	216	0.047	35	0.067
7	760	0.44	108	0.095
8	2725	0.051	369	0.160
9	9910	0.070	1285	0.39
10	36446	0.13	4655	1.21
11	135268	0.24	17073	4.6
12	505861	0.74	63600	18.2
13	1903890	2.5	238591	79.4
14	7204874	11.33	901971	?

Task 4

We implemented the naive algorithm given to solve the exact cover problem on various instances. We then ran tests on two of these instances, namely :

– Instance 1 :$X = \{1, 2, ..., n\}, S = \mathcal{P}(X)$
– Instance 2 :$X = \{1, 2, ..., n\}, S = \{U \subset X, |U| = 2\}$

n	Instance 1	Execution time (s)	Instance 2	Execution time (s)
5	52	0.0033	n/a	-
6	203	0.010	15	0.00099
7	877	0.033	n/a	-
8	4140	0.11	105	0.0074
9	21147	0.59	n/a	-
10	115975	3.53	954	0.044
11	678570	33.55	n/a	-
12	4213597	12 min	10395	0.33
13	27644437	40 min	n/a	-

On the other hand using the heuristic suggested when choosing the random element of X we get the following performances:

n	Instance 1	Execution time (s)	Instance 2	Execution time (s)
5	52	0.0028	n/a	-
6	203	0.0092	15	0.0011
7	877	0.056	n/a	-
8	4140	0.15	105	0.0076
9	21147	0.54	n/a	-
10	115975	3.95	954	0.045
11	678570	35.9	n/a	-
12	4213597	12 min	10395	0.33
13	27644437	40 min	n/a	-

A quick comparison shows us that the heuristic fails to enhance the execution time of the algorithm. The problem is that when looking for the element that appears the fewest time we use a loop with linear complexity and this is repeated as long as the set X hasn't been exhausted yet, moreover although our algorithm does generate less branches overall, the fact that our exact cover instances are very symmetrical makes the gain in branches too little to compensate the cost of our linear loop search.

What would be interesting is to implement a data structure that stores the least appearing element and updates it with each execution so that we can reach it faster.

Task 5

We implemented the dancing links data structure using 3 separate classes:

– **Data** : Codes the data objects, and has the 5 required attributes to which we added a sixth one for code optimization purposes.

  - **U**: Points the upward data.
  - **D**: Points the downward data.
  - **R**: Points the rightward data.
  - **L**: Point the leftward data.
  - **C**: Points to the corresponding column object.
  - **set**: Stores the subset index that the data object belongs to as an integer object.
- **Column** : Codes the column data objects. We construct this class using inheritance from the **Data** class and add the missing attributes:
  - **S** : Stores the number of 1's in the column.
  - **N** : Stores the name of the column. For simplifications we implemented it as an integer object.
- **DancingLinks**: Is the main class where the algorithm is implemented. We defined a **DancingLinks** object by its **H** column since it lets us access the whole dancing links structure.

Task 6

We used the DancingLinks structure to optimize the exact cover algorithm, we then ran tests which gave the following results:

n	Instance 1	Execution time (s)	Instance 2	Execution time (s)
5	52	0.00043	n/a	-
6	203	0.00137	15	0.00021
7	877	0.00425	n/a	-
8	4140	0.0056	105	0.00053
9	21147	0.022	n/a	-
10	115975	0.10	954	0.0022
11	678570	0.60	n/a	-
12	4213597	2.88	10395	0.011
13	27644437	37.99	n/a	-

Task 7

In order to convert a simple tiling problem with a **GroundSet** $\mathcal{G}$ and a **Tiles** set T we implemented hashing methods to represent the different polyominos as lists of integers from which we could reconstruct those polyominos in linear time with respect to their size.

We use a simple hash function to code a **Point** object, more precisely, given a **Key** n (sufficiently big) we code the **Point(x,y)** as $n.(y + \frac{n}{2}) + x + \frac{n}{2}$.

The reason behind adding the $\frac{n}{2}$ is to force the positivity of terms, and since in practice $n >> x, y$ we can easily compute x, y from the euclidean division of the code over the **Key** n. We can thus represent each polyomino in $\mathcal{T}$ as a

integer subset and we can represent $\mathcal{G}$ as a set X of integers representing the unit squares forming **GroundSet**.

The conversion of a tiling problem with possible repetitions into a exact cover one is quite straightforward as explained above.However in order to force our tilings to us each polyomino "type" at most once we need to modify our algorithms and implement this constraint somehow in the dancing links data structure.

The general idea is to add "fictional" columns in our exact cover matrix that code these constraints, in fact the latter idea is the the essence of the various applications of exact cover algorithms to more practical problems (Sudoku, N-queen problem, Perfect matchings...).

More precisely, let $Tiles = \{Polyo_{i,j}, i \in \mathcal{A}_j, j \in \mathcal{C}\}$ be the set of tiles where $Polyo_{i,j}$ is a polyomino belonging to the class j, and appears $Card(\mathcal{A}_j)$ times in $Tiles$. We construct our exact cover matrix structure as follow:

- Build the classical representation for the tiling problem with repetitions.
- For each $j \in \mathcal{C}$ add a column $Column_j$ at the end (to the right) of the other columns.
- For each $j \in \mathcal{C}$,$i \in \mathcal{A}_j$, put a 1 in the corresponding row of $Column_j$, and 0 in each other corresponding row of $Column_{j'}, j' \neq j$.

It's quite easy to see that if a solution exists it will use exactly once each "type" of polyomino.

Task 8

Sub Task a

In this sub-Task we used our tiling algorithms to look for possible tilings of the given figures using each pentamino exactly once, we call these figures $\mathcal{F}_a, \mathcal{F}_b, \mathcal{F}_c$ from left to right. We found the following results:

- $\mathcal{F}_a$: There are 404 possible tilings,
- $\mathcal{F}_b$: There are 374 possible tilings,
- $\mathcal{F}_c$: There are 0 possible tilings,

Sub Task b

We considered two instances of rectangle tiling problems:

- Tiling a 6×3 rectangle by k-size fixed polyominos.
- Tiling a $n \times n$ square by k-size fixed polyominos.

k	Nbr of tilings 2nd case
1	1
2	41
3	170
4	0
5	0
6	132

k,n	Nbr of tilings 1st case
1	1
2	2
3	10
4	117
5	4006
6	451206

Sub Task c

We ran programs that tested each octomino and tried to find a tiling of its 4-Dilate. We found exactly 10 free octominos satisfying this condition, and what is quite remarkable is that all these octominos have the property of forming small rectangular blocs when 2 of them are combined.

Extensions : Sudoku

The idea used here to transform a Sudoku problem into an exact cover problem is the same used in **Task 8** to find tilings using each polyomino exactly once. A soduku is a 9×9 grid with integer entries between 1 and 9, moreover 3 constraint must be satisfied by the grid:

- **Row-Column** : Each intersection of a row and a column contains exactly 1 value between 1 and 9.
- **Row-Value** : Each row must contain each value between 1 and 9 exatly once.
- **Column-Value**: Each column must contain each value between 1 and 9 exatly once.
- **Box-value** : Each of the 9 principal 3×3 sub grids of the Sudoku contains each value between 1 and 9 exactly once.

These constraint are added into the exact cover data structure by considering the classical subsets S_i as triplets $(Row, Column, Value)$ and the ground set X as the combination of constraints. We strongly recommend the example given Here of such a matrix for a 4×4 Sudoku variant as it is more comprehensive than many stacked paragraphs.

References

1. Don Hugh Redelmeier. *Counting Polyominos : Yet Another Attack.* (English) . Discrete Mathematics 36 (1981) 191-203.
2. Donald Ervin Knuth. *Dancing Links.* (English) . Millenial Perspectives in Computer Science, 2000, 187–214.

Annex (Figures)

Fig. 1: Task 1 - Polyominoes from the file `polyominoesINF421.txt`

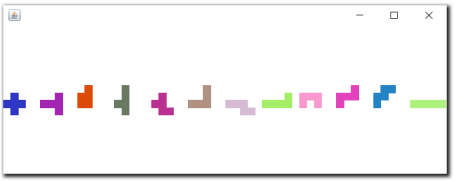


Fig. 2: Task 2 - The 12 free pentaminos

Fig. 3: Task 2 - The 63 fixed pentaminos

Table 1: Task 8 - Tiling rectangles

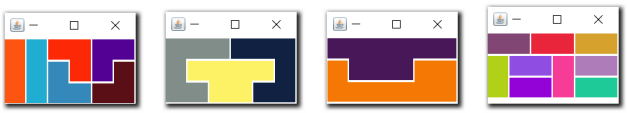
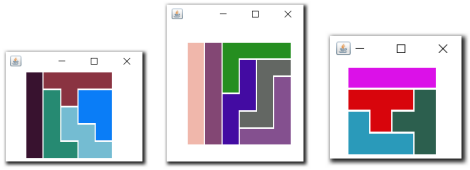


Table 2: Task 8 - Tiling squares



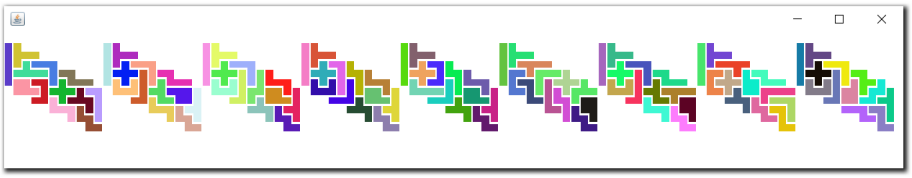


Fig. 4: Some exact tilings of $\mathcal{F}_a$

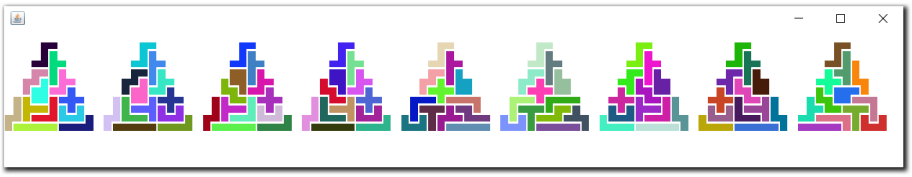


Fig. 5: Some exact tilings of $\mathcal{F}_b$

Fig. 6: The 10 octominos that cover their 4-dilate

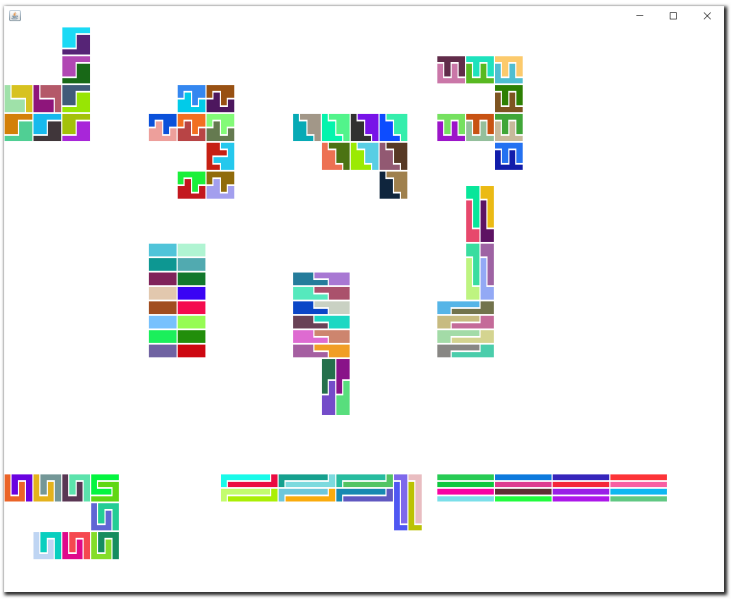


Fig. 7: Some of the tilings generated

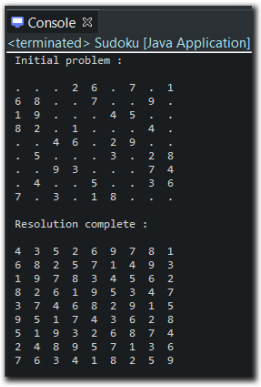


Fig. 8: Resolution of a Sudoku problem