INF421 PROGRAMMING PROJECT
POLYOMINO TILINGS AND EXACT COVER

Luyi SHEN & Cheng ZHANG

1. Polyominoes

Task 1. *Implement a representation of polyominoes supporting the following functionalities:*

- *creation of a polyomino from a string and creation of a list of polyominoes from a text file;*
- *drawing;*
- *elementary transformations such as translations, rotations (quarter-turn around the origin), reflections (with respect to a coordinate axis), and dilations;*

For the representation of polyominoes:

- we have firstly wrote a class called "*Coords_rep*" representing every unit of square because every polyomino is consisted of a set of squares. In the class "*Coords_rep*", there are three members which represent respectively the origin (the left-bottom coordinate of the square), the x-coordinates and the y-coordinates which are used for drawing afterwards.
- Then we have wrote the class "*Polynomio*" which contains a member which is a LinkedList of *Coords_rep* for representing the polyominoes.

For the creation of a polyomino from a string, we have wrote a constructor *public Polynomio(String S, int ox, int oy)* in which the String S contains the origin coordinates of all the squares of this polyomino. For example, with the String *s = "[(0,0), (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (2,1), (2,2), (3,0), (3,1), (3,2), (3,3), (3,4)]"*, we can obtain the polyomino like Figure 1-1.

Figure 1-1: the image of the polyomino obtained by String s

For the creation of a polyomino from a text file, we have wrote a function *public static LinkedList<Polynomio> toList(File file)*. For example, with the given file *polyominoesINF421.txt*, we obtain the polyomino like Figure 1-2.

Figure 1-2: the image of the polyomino given in the text file *polyominoesINF421.txt*

For drawing, we use the given class "*Image2d*" and we write a function *public Image2d toImage(Image2d img)*.

For the transformations like translations, rotations, reflections and dilations, we write four functions *Translation(int i, int j)*, *rotate(int deg)*, *reflection(int axis)* and *Dilate(int r)* in the class "*Polynomio*" and four functions *Translation(int i, int j)*, *rotate(int ox, int oy, int deg)*, *reflection(int ox, int oy, int axis)* and *Dilate(int ox, int oy, int r)* in the class "*Coords_rep*". For the rotation, the parameter *deg* can be 1, 2 and 3 which represent 90, 180 and 270 degrees. For the reflection, the parameter *axis* can also take four values which represent the reflection with respect to the x-axis, y-axis, the ascending diagonal and the descending diagonal. Take the String s as example, after the four transformations, we obtain the polyominoes like Figure 1-3.

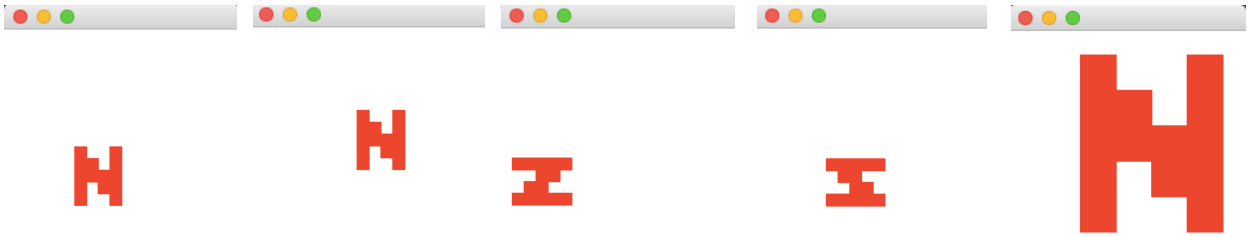

Figure 1-3: The polyominoes after the four transformations. (a)original polyomino; (b)after the translation to right-up; (c)after the counterclockwise rotation of 90 degree; (d)after the reflection with respect to the ascending diagonal; (e)after the dilation of 3

Task 2. *Write methods that generate all fixed polyominoes and all free polyominoes. How far can you push these enumerations?*

In order to generate all fixed polyominoes, we write a function *public static void enumerate_naive(LinkedList<Polynomio> list, Polynomio init, int ox, int oy, int P)* which enumerates all fixed polyominoes with size ≤ P (maybe repeated) and a function *public static LinkedList<Polynomio> generate_fix_naive(LinkedList<Polynomio> list, int size)* to get the polyominoes of size *size* not repeated. The pseudocode of the function *enumerate_naive* is:

```
if the size of init ≤ P , do
        add all squares in polyomino init to visited
        for each square u in visited
                for each neighbor c of u which is not in visited
                        add this neighbor c to the original polyomino init
                        add this new polyomino to list
                        call recursively enumerate_naive with the new list and the new init
                        remove the last element of init
                end for
        end for
end if
```

And in the function *generate_fix_naive*, we select the polyominoes of a certain size and test if a polyomino is already appeared. If yes, we do not add this polyomino to the final list.

In order to generate all free polyominoes, we write a function *public static LinkedList<Polynomio> generate_free(LinkedList<Polynomio> Poly_fix)* and an auxiliary function *public LinkedList<Polynomio> all_trans()* which returns all rotations and reflections of a polyomino. So in the *generate_free*, we test if some rotations and reflections of a polyomino are already existed in the list. If yes, we do not add this polyomino to the final list.

We can push these enumerations until size 7.

Task 3. *Implement this method (the method of D. H. Redelmeier) to generate further all fixed polyominoes and all free polyominoes. Compare to your solution.*

We have used the method of D. H. Redelmeier to generate all fixed polyominoes by writing a function *public static void enumerate(LinkedList<Polynomio> list, LinkedList<Coords_rep> neighbours, Polynomio parent, LinkedList<Coords_rep> untried, int ox, int oy, int P)* and we can push the enumerations of size 13 in a few minutes. It is much better than our naive method because it generates directly all fixed polyominoes without repeat instead of testing and removing the repeated polyominoes. However, we failed to implement the method of generating all free polyominoes, so we still use our naive method.

2. Polyomino tilings and the exact cover problem

Task 4. *Implement this algorithm to solve the exact cover problem. Test your algorithm on the exact cover problem (1) as well as on large instances (such as all subsets, or all subsets of size k of a ground set with n elements). How far does your algorithm solve the exact cover problem?*

To implement this algorithm, we have wrote a class called "*Exact_Cover*" with two fields: *Vector<Integer> X* and *LinkedList<Vector<Integer>> C* which represent respectively the ground set X and the collection C of subsets of X. And in this class, we write a function *public LinkedList<LinkedList<Vector<Integer>>> exactCover()* to solve the exact cover problem. It returns all possible solutions in the forme of a linked list.

For the problem (1), the algorithm returns the solution *{ { 3 5 6 } { 2 7 } { 1 4 } }* and we also test the algorithm on different instances like {{1}{2}{3}{4}{5}{6}} to test the situation in which we have no solution, also all the cases of all subsets, and our algorithm returns all correct solutions.

Task 5. *Implement the dancing links data structure, and write code that transforms an exact cover problem to its corresponding dancing links data structure.*

In this task, we have wrote three classes called "*Data*", "*Column*" and "*Dancing_Links*".

- In the class "*Data*", there are five members *Data U, D, L, R* and *Column C* which represent the up, down, left and right of the present *Data* and the *Column* to which this *Data* is belonging.
- In the class "*Column*" which is a special *Data* object, so it heritage Data , but it has two members supplementary: *int S* and *String N* which represent the size of this *Column* (the number of *Data* which belongs to this *Column*) and the name of this *Column*.
- In the class "*Dancing_Links*", there is a member called *H* of type *Column* which is the head object of a dancing links data structure. And we write a constructor *public Dancing_Links(int [][]M)* which transforms a matrix M to dancing links.

And in the class "*Exact_Cover*", we add two functions *public int[][] toMatrix()* and *public Dancing_Links toDL()* which transform an exact cover problem to its corresponding matrix and to its corresponding dancing links.

Task 6. *Implement the dancing links algorithm to solve the exact cover problem. Test your implementation on the exact cover problem (1) as well as on large instances (such as all subsets, or all subsets of size k of a ground set with n elements). How far does your implementation solve the exact cover problem?*

In order to implement the dancing links algorithm, firstly we write two functions *public void coverColumn()* and *public void uncoverColumn()* in the class "*Column*". And then write a function *public PriorityQueue<Column> toPQ()* in the class "*Dancing_Links*" which stock all columns in a dancing links data structure to a priority queue in order to easily get the column with the minimal size. Finally, we write the function *public LinkedList<LinkedList<Vector<Integer>>> exactCover(PriorityQueue<Column> pc)* to solve the exact cover problem using the dancing links algorithm.

We have tested our code on several instances and we compare the time used in two algorithms. We find that the dancing links algorithm is much faster than the method of task 4. For example, for a 35×5 matrix, the dancing links algorithm takes 2 time units and the method of task 4 takes 14 time units. And we observe that the number of columns has very little impact on the time consumed to solve the exact cover problem. But when the number of rows of the dancing_links raises to several hundreds, it takes more than half an hour to solve the problem.

Task 7. *Observe that the problem of tiling a polyomino P using some polyominoes of a set S with possible repetitions can be represented by an exact cover problem where the ground set is the collection of all squares of P, and the subsets correspond to the squares of P covered by a polyomino of S placed at a certain position. Explain how to adapt this representation so that each polyomino of S is used exactly once.*

In order to use the P with possible repetitions, we can create the matrix of the exact_cover problem as below.

- In the ground set X, there will be all unites of the polyomino P to cover.
- In each subsets of C, we represent all possible positions of the polyominoes in S by marking 1 below the corresponding unites.

More precisely, we let each element in the ground set X as the representation of the columns of the matrix M and each subset of C as a row of M in which we set a 1 if this subset covers a certain element of X. And now, to guarantee that each polyomino of S is used exactly once, we add some columns in M which represent each polyomino of S and in each row, we set a 1 to the polyomino that this subset represents.

Task 8. *Implement a method that transforms a polyomino tiling problem (allowing or not rotations, symmetries and reusable tiles) to an exact cover problem. Apply for example to solve the following polyomino tiling questions:*

- *Find all tilings of the polyominoes of Figure 2-1 by all free pentaminoes.*
- *For a given n (say n = 4, 5, or even maybe 6), find all tilings of a rectangle by all fixed (resp. free, resp. one-sided) polyominoes of area n.*
- *For a given n and k, find all polyominoes P of size n which can cover their own dilate kP. How many do you find for (n; k) = (8; 4)?*

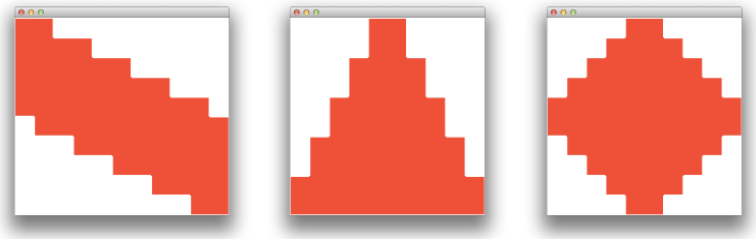

Figure 2-1: Find all tilings of these polyominoes by all free pentaminoes

In this task, we use the dancing links algorithm to reduce the calculating time. For every example, we write one different function in the class "*Polynomio*".

For the first example, we have wrote a function *public static LinkedList<LinkedList<Polynomio>> exactCover1(LinkedList<Coords_rep> X, LinkedList<Polynomio> C, int oy)* in which X are the polyominoes in Figure 2-1 and C is the collection of all free polyominoes of size 5. Unfortunately, our algorithm takes too much time to get the solutions because the corresponding matrix has too large size. We do not get the results.

For the second example, we write two functions *public static LinkedList<LinkedList<Polynomio>> exactCover2_fix(int n, LinkedList<Coords_rep> X, int ox, int oy)* and *public static LinkedList<LinkedList<Polynomio>> exactCover2_free(int n, LinkedList<Coords_rep> X, int ox, int oy)* to solve respectively two cases of fixed polyominoes and free polyominoes. we define a 4×6 rectangle for n = 4, a 5×7 rectangle for n = 5 and we get all possible solutions in a few seconds.

For the third example, we use the function *LinkedList<LinkedList<Polynomio>> exactCover1(LinkedList<Coords_rep> X, LinkedList<Polynomio> C, int oy)* and we iterate all the free polyominoes of size 8, and X is the free polyomino and C is the collection of its transformation. We get (8,4) = 10. And Figure 2-2 shows these 10 self-cover polyominoes.

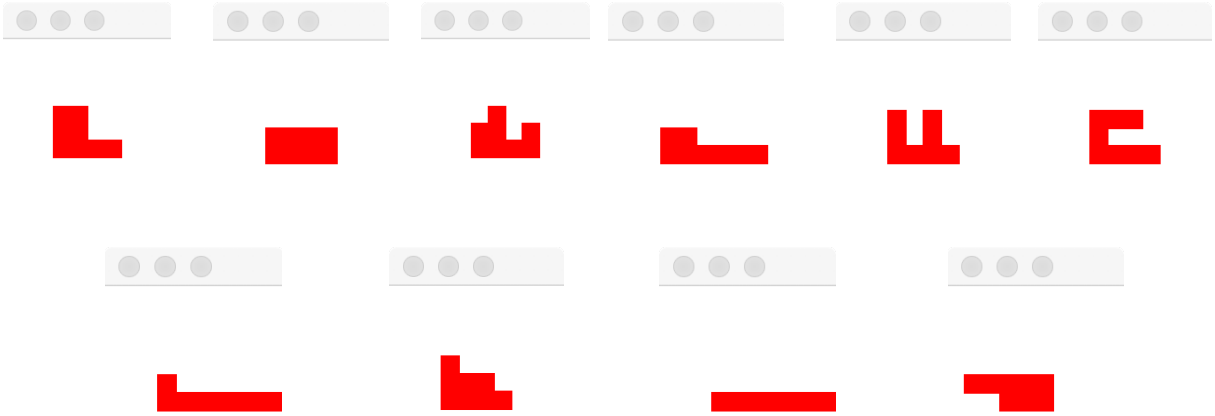

Figure 2-2: All self_cover polyominoes of size 8

3. Extensions

Task 11. *Observe that a sudoku grid can be seen as a solution to an exact cover problem. What size do you need for the ground set X and the collection C of subsets of X? Implement a sudoku solver using your dancing links algorithm.*

We have chosen the sudoku solver problem to implement our algorithm for exact cover problem. We can transform a sudoku problem like this:

- we need to fill in every case;
- we need to write the number 1~9 in every row;
- we need to write the number 1~9 in every column;
- we need to write the number 1~9 in every small grid.

The first condition need 81 elements because we have 81 cases. Each remaining condition needs 81 elements because we have 9 rows (columns and small grids) and each row (column and grid) has 9 numbers to fill in. So for the ground set X, we need 324 as its size.

When it comes to the construction of the collection C, we can say that for each case, if there is already a number, it becomes one subset and we transform the case and the number to its corresponding conditions. If there is no number in this case, it becomes nine subsets which represent the nine different cases when we write the number 1~9. So the size of C change for different sudoku problems, but it will not be greater than 729.

We write a class called "*sudoku*" to solve the sudoku problem. It contains two principal functions *public static int[][] toMatrix(String s)* and *public static void afficher(LinkedList<LinkedList<Vector<Integer>>> l)*. The first function transforms a String *s* to the corresponding matrix of the exact cover problem. The String contains all the cases with its number in it. The second function transforms the results of exact cover problem to the results of sudoku problem. And in the *main* function, we call the dancing links algorithm.

For example, the solution of the sudoku problem of Figure 3-1 is *{ {(7,9),5} {(9,9),3} {(2,9),6} {(2,8),3} {(7,1),3} {(8,5),3} {(9,1),6} {(2,4),2} {(8,8),6} {(7,4),1} {(6,8),7} {(8,1),5} {(9,7),4} {(7,7),9} {(2,5),7} {(9,3),9} {(4,7),3} {(3,5),6} {(3,6),1} {(2,6),8} {(9,2),2} {(8,6),2} {(4,6),7} {(5,8),8} {(6,2),5} {(3,7),7} {(7,5),4} {(7,3),7} {(4,8),5} {(8,2),1} {(1,8),4} {(3,3),5} {(5,2),7} {(1,9),8} {(5,5),5} {(2,2),4} {(8,4),9} {(1,3),2} {(3,9),2} {(5,7),2} {(1,7),1} {(3,1),8} {(1,1),7} {(4,2),9} {(5,3),6} {(6,6),4} {(6,3),3} {(6,4),8} {(9,8),1} {(9,6),5} {(4,4),6} {(9,5),8}*

{(9,4),7} {(8,9),7} {(8,7),8} {(8,3),4} {(7,8),2} {(7,6),6} {(7,2),8} {(6,9),9} {(6,7),6} {(6,5),1} {(6,1),2} {(5,9),1} {(5,6),9} {(5,4),3} {(5,1),4} {(4,9),4} {(4,5),2} {(4,3),8} {(4,1),1} {(3,8),9} {(3,4),4} {(3,2),3} {(2,7),5} {(2,3),1} {(2,1),9} {(1,6),3} {(1,5),9} {(1,4),5} {(1,2),6} }. The coordinate is defined as: the left-up case *init* is (1,1), the right case of *init* is (1,2), the down case of *init* is (2,1). The rest can be done in the same manner.

	6		5	9	3			
9		1				5		
	3		4				9	
1		8		2				4
4			3		9			1
2				1		6		9
	8				6		2	
		4				8		7
			7	8	5		1	

Figure 3-1: A sudoku problem