INF421 – Programming project

Polyomino tilings and exact cover

LÂASRI Julien

SUN Wang

1. Polyominoes

Manipulate polyominoes

Task 1

First of all, we should choose a class to represent the polyominoes. We need to read a string or the data from a file to generate polyominoes. In the following part, we need to construct the polyominoes from different data forms so that the constructor is overloading. For generating a list of polyominoes from a text file, we use a linked list to store them. In the part of polyomino tilings, we need to draw the polyominoes out, thus we use a function “toPolygon” to transform each polyomino into a list of Polygons which belong to one package of Java, and then draw these Polygons on screen. In this function, we deal with the anti-direction of axis y of Java against our conventional direction for the direct view of transformations. Among four elementary transformations, rotation, reflection and dilation all have two steps. First, we consider the region where the polyomino is and choose the point (minx, miny) which is the corner of the region as origin of operation. Second, we translate the polyomino after the operation to the original place.

Generate polyominoes

There are two ways to generate polyominoes. The first way is to generate them from existing data, which is described in the previous part. The second way is to generate polyominoes given only the size and the symmetry demanded.

Task 2

We considered polyomino as a graph and generated it with the way depth-first search. We started from the origin point (0, 0), and recurred to one of his neighbours until size p that we set. In fact, our idea is similar to the algorithm given by the article of D.H.Redelmeier, especially for fixed polyomino. And for free polyomino, we chose the way of deleting the duplicates for each symmetry. For one-sided, we employed the method of D.H.Redelmeier. The enumerating is not as fast as the algorithm of D.H.Redelmeier. Thus, we modified our program and the result: for fixed, we tested until 16; for free, until 10.

Task 3

The article of D.H.Redelmeier concentrates on the different symmetries of polyomino. But first of all, we focus on fixed polyomino. The structure of generating polyomino is like a “family tree” as it uses the depth-first search. And each child polyomino in the tree consists of its parent plus one new neighbour point. Because of the frequent operation of the set of points that is not tried, we chose a linked list to store them. With this linked list, we will not use a point repeatedly and avoid counting the same form two or more times. Once we choose a new point to generate a child polyomino, we should add the neighbours of this new point into account. Every time entering the recursion, the function takes child polyomino as its parent polyomino and copies the untried linked list as its linked list. Now we set polyomino parent, the linked list of points untried and the limit of size P as parameters of the function count. So the approach is as follows:

```
count(parent, untried, P)=
while(untried is not exhausted) do:
  if parent == null then do:
    remove m from untried;
```

```
          use m to form child polyomino;
        if the size is less than P then do:
          add the neighbours of m to untried;
          copy untried;
          count(child, copied untried, P);
            remove the neighbours of m from untried;
          else do:
            number of polyomino with size P plus one;
          remove m;
        else do:
          remove m from untried;
          use m and points of parent to form child polyomino;
        if the size is less than P then do:
          add the neighbours of m to untried;
          copy untried;
          count(child, copied untried, P);
            remove the neighbours of m from untried;
          else do:
            number of polyomino with size P plus one;
          remove m;
```

If we need not only the number of polyomino with size P but also the polyominoes themselves, we can add a linked list of polyomino to store every time the number augments.

The symmetries are divided into simple and composite. The algorithm given by the article enumerates each symmetry individually. But for each symmetry, there is a difference between the positions of symmetries. Either between the points, either across the points. It employs the character of symmetry to simplify the program like counting the half of the horizontal symmetry with axis between the points, and changing the x-axis to the diagonal axis for the ascending diagonal symmetry. It changes the way of counting to adjust to the horizontal symmetry with axis across the points. The vertical symmetry is just the same as the horizontal symmetry except the direction. And descending diagonal symmetry also. For the rotational symmetry, it employs an idea of ring to generate. Unfortunately, we have difficult in generating the ring correctly. Hence, we use the fixed polyominoes and delete the duplicates in the file MonR, and the speed of enumerating is not so good.

For fixed, we tested until 16; for none, tested until 16; for all, we tested until 25; for axis2, we tested until 25; for rotate2, we tested until 10; for diag2, we tested until 25; for axis, we tested until 19; for rotate, we tested until 14; for diag, we tested until 19. All the results above are correct according to the data in the article of D.H.Redelmeier. For the size resting to test, we used the function axisp which represents at least symmetry axis to test the sum of all+2axis2+2axis. And the result is correct until 25, we cannot test more. According to this, if we test axis until 25, it will also be correct. For diagp=all+2diag2+2diag, it’s the same situation.

2. Polyomino tilings and the exact cover problem

The exact cover problem

The algorithm chosen to solve the exact cover problem is pretty intuitive and it does not really change throughout the pdf of the project. In fact, what changes is only the data structure we use to solve the problem, and this drastically affects the effectiveness of our program and this is where almost all the optimization takes place.

The algorithms of the tasks 4, 5 and 6 were pretty much described in the document, however the challenging part of the implementation was how to switch from a certain data structure to another. That is what we're going to analyse here.

Task 4

In order to switch from an exact cover problem represented by a matrix m, to one represented by sets, we first create the set X containing all columns of m. Then, to create all the subsets of C, for each line i of m, which represents a subset S of X, we check all elements m[i][j] for all j possible. If m[i][j] = 1, it means the element represented by the column j is in the subset S (represented by i), so we add this element to S. We thus obtain X and C and we execute the algorithm we already have with sets to solve the problem.

Task 5

We now have to transform an exact cover problem given by a binary matrix m of integers (0 or 1) to the same problem in its dancing links data structure form, which means we have to return the corresponding column H which is linked with all other 'cells' (the Data). We proceed in different steps:

- We create the column H linked horizontally to itself.
- We convert m into a matrix m2 of Data where if m[i][j] = 1, m2[i][j] contains a Data structure linked horizontally to itself.
- Then, we create the columns one by one. We go through the matrix m2 from the upmost-leftmost corner to the downmost-rightmost corner. Thus, when we encounter a not-null Data when creating the column, we only have to check for neighbours on the right and under it in order to update all fields L, R, U and D.

Task 6

To compare the two implementations of the algorithm, we use a function that, for a given n, creates the exact cover problem for a set of size n we try to cover with its subsets. We thus obtain the comparison table below of the time taken to solve this exact cover problem for different values of n.

	Time **without** dancing links (milliseconds)	Time **with** dancing links (milliseconds)
n = 8	74	7
n = 9	552	9
n = 10	3324	139
n = 11	30932	573

We see how the algorithm using dancing links becomes more effective pretty fast.

Polyomino tilings

Task 7

As mentioned in the pdf, 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. If we want to adapt this representation so that each polyomino of S is used exactly once, we need to add new elements to the ground set for each polyomino. This new element should be common to all transformations of a polyomino that are supposed to be representing the same polyomino in the given circumstances (its rotations, translations or symmetries). Thus, in a given cover, this element cannot be present in two different subsets, and should be present in at least one of them, this basically means each polyomino (including its rotations, translations and symmetries) will be used exactly once.

Task 8

The program that solves the tiling problem is divided into three parts:

1. The translation of the tiling problem into an exact cover problem represented by a matrix m

In order to translate our tiling problem into a matrix m which represents an exact cover problem, we need to attribute a unique id to all elements of the polyomino P we try to cover. We store this unique id in an array as we will need it further in the program. For each polyomino of the list l that can be used to cover P, we then create a set polyominoAssociated which contains its rotations (if we allow rotations) and symmetries (if we allow symmetries). Then, we create the minimum rectangle that contains P (represented by P.binaryMatrix( )) and for each point p in this rectangle, we create a unique id for each polyomino q in polyominoAssociated placed at this position. If it actually fits at this position in the polyomino P, we then add its representation in the matrix using the unique Ids we have created for itself and for the points that are contained in it. If we want to use all polyominoes exactly once, we add the extra element of the ground set corresponding to the polyomino common for all elements of polyominoAssociated, as said before.

2. We solve the exact cover problem associated

We obtain the solutions of the exact cover problem associated with this polyomino tiling problem by running our previous program using dancing links data structure.

3. We build the polyomino tiling problem from the previous results

For each solution of the exact cover problem, we build each polyomino square by square by assembling all squares of the same subset. Here we use our storage of the unique id given to all elements of the polyomino P we try to cover to get the Point (and thus the square) the given id corresponds to.

We implemented other functions to test our code, we can thus draw the solutions of these types of polyomino tiling problems. You can find an example of the drawing of solutions with the image “results/All404CoversFigure5aPentominoes.png” where we draw all 404 possible polyomino covers of the first polyomino given in figure 5 by all pentominoes.