Ecole Polytechnique

INFO 421: Design and Analysis of Algorithms

Polyomino Tilings and Exact Cover Problems

Joshua Benabou

1 Manipulating polyominos

In order to represent and manipulate polyominos,we first define a Square object representing a square of the integer lattice by it's bottom-left point (i, j), and we impose a total order on the set of Squares. A Polyomino object is then defined as an ArrayList of Squares.

By operating on each Square in the polyomino's list of vertices we can easily write methods for manipulating polyominos, e.g translation, reflection across the x or y-axes, rotation by $\pi/2$, dilation by an integer factor k and also copying, constructing polyominos from a text file of vertex lists, and displaying, using the graphical interface Image2D (which we completed).

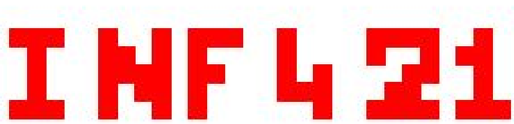

Fig 1: polyominos extracted from polyominoesINF421.txt

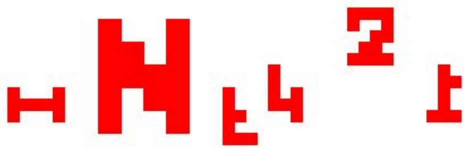

Fig 2: the same polyominos after elementary transformations

2 Generating polyominos: naive approach

In order to generate polyominos of a given area n and type $\mathcal{T}$ (fixed, onesided, or free), we first try a naive inductive approach. In order to not distinguish between translations, we define a canonical form for each polyomino P in which the lattice square $(x_{\min}, y_{\min})$ is translated to the origin, where $x_{\min}$ (resp. $y_{\min}$) is the minimum of the x (resp. y) coordinates of the vertices of P.

We have incorporated into the Polyomino constructor a line for putting it into canonical form. Any time we perform a transformation on a polyomino we have to take care to put it back into canonical form. We generate polyominos in this canonical form according to algorithm (1):

Result: List of polyominos of type-$\mathcal{T}$, area n
If $n = 1$, return list of one element, the Polyomino with one square at the origin;
Create empty list L;
for *polyomino P of type $\mathcal{T}$ and area $n-1$* **do**
  for *squares s of P* **do**
    for *neighbors a of s* **do**
      if $a \notin P$ **then**
        Define new Polyomino P' such that $P'.vertices = P.vertices \cup \{a\}$
        if *L contains no $\mathcal{T}$ - symmetry of P'* **then**
          $L = L \cup \{P'\}$
        end
      end
    end
  end
end

We define a $\mathcal{T}$-symmetry of a polyomino P as follows: for fixed, simply P itself; for onesided, the set of rotations of P; for free, the set of (at most 8) symmetries of P.

In order to test whether a newly generated polyomino P' is already in our list L, we must be able to test the equality of two polyominos. As we must do this many times, we have chosen to work with polyominos in which the vertex list is ordered; we create a new polyomino P' by inserting a new square a in the vertex list of P, such that the vertex list remains ordered. An equality test of two polyominos in which the smallest has area n can then be done in $O(n\log(n))$.

To efficiently check whether a candidate square a is already in P, we define some sufficiently large rectangle R containing P, and store in an array the truth values of whether the squares of R are in P or not. Hence we can check if $a \in P$ in $O(1)$.

To study the time complexity of algorithm (1), consider the case of fixed polyominos. Let there be P_n of them of area n. Consider the generation of poloymonos of area $n+1$ (complexity C_{n+1}). We end up creating $M = (n+1)P_{n+1}$ new polyomino objects P' because for each of the P_{n+1} polyominos of area $(n+1)$, there are $(n+1)$ ways to create it from a polyomino of order n by adding one square. To create such a P' we clone a P of area n in $O(n)$ and insert a into an ordered vertex list in $O(n)$. Polyomino creation thus contributes $O(Mn)$.

We then check for equivalence between polyominos at most MP_{n+1} times (loose bound), each time in $O(n)$ as the vertex lists are ordered. In fact, equality testing is the dominant term for the complexity of each recursive step (proof omitted). Hence: $C_{n+1} \leq C_n + O(n^2 P_{n+1}^2)$. It is known that the number of polyominos grows exponentially, so the complexity is exponential in n.

3 Redelmeier's algorithm

We can generate fixed polyominos more intelligently (and quickly!) by avoiding equality tests between polyominos using Redelmeier's algorithm, which generates all fixed polymominos up to a given area.

Redelmeier defines a canonical form in which the bottom leftmost square is placed at the origin. We have thus overided our original Polyomino constructor to avoid translating to the origin.

To obtain free (resp. onesided) polyominos, we select only those fixed polyominos which are the "minimal" member of their symmetry (resp. rotation) group."Minimal" is in the sense of the lexicographic order defined for Polyominos, which is induced by total order defined for Squares.

Fig 3: comparison of runtime performances for naive and Redelmeier algorithm

#Polyominos			Runtimes (ms)			
n	#Fixed	# Free	Algo1 Fixed	Algo1 Free	Redelmeier Fixed	Redelmeier Free
5	63	12	5	4	1	9
6	216	35	16	15	2	49
7	760	108	177	34	3	82
8	2725	369	730	277	8	247
9	9910	1285	6620	1131	36	495
10	36446	4655	116746	7526	149	1815
11	135268	17073	?	166648	749	7354
12	505,861	63600	?	?	4641	27723
13	1903890	238591	?	?	12374	129830

4 Exact covers: naive backtracking solution

To easily convert between exact-cover representations we wrote a method outputting the matrix $M(X, C)$ given the sets (X, C). To implement the naive backtracking solution to the exact cover problem we make extensive use of the Java data structure Set; an exact cover is a set of sets of integers.

When choosing the element $x \in X$ to cover first, we have the option of implementing the following heuristic limiting branches at each step: choose x such that the number of $S \in C$ with $x \in S$ is minimal.

We tested the algorithm on the following two exact-cover problems, with and without the above heuristic, which did not change runtimes significantly.

Problem 1: $X = \{1, \ldots, n\}, C = \mathbb{P}(X)$

Problem 2: $X = \{1, \ldots, n\}, C = \{S \in \mathbb{P}(X), |S| = k\}$

Runtime performance and comparison with the smarter DL algorithm are given in section 6.

5 Dancing Links data structure

To implement the dancing links data structure, we first define a *data_object* object, which contains the fields C, U, D, L, R and as well as an integer row identifier *row_id* (not necessary, only used for the Sudoku solver, see [9]). Instead of defining a different *column_object* object we will simply add two fields to the *data_object* and define them only when needed: an integer column id *col_id* and the size of the column.

To transform a $r \times c$ exact cover matrix M to its corresponding dancing links data structure, we define a linked $(r + 1) \times c$ *data_object* matrix in which the last row represents the column headers. We first initialize the cyclic link structures amongst the column headers. Next, for each entry e of M equal to one, we update the size of the corresponding column header, and we look for the closest entries equal to one in the directions U, D, L, R to create 4 links to e (note that e could link to itself).

6 Solving exact covers with Dancing Links

Fig 4: Runtime performance of naive backtracking and DL for the exact cover problems presented in section 4.

Problem 1				Problem 2			
n	#covers	Naive (ms)	DL (ms)	(n, k)	#covers	Naive (ms)	DL (ms)
5	52	7	1	$(8, 2)$	105	18	2
6	203	20	2	$(8, 4)$	35	14	2
7	877	94	6	$(9, 3)$	280	91	5
8	4140	1131	21	$(10, 2)$	945	362	11
9	21147	23896	85	$(10, 5)$	126	187	8
10	115975	?	383	$(12, 3)$	15400	56979	129
11	678570	?	4233	$(12, 4)$	5775	9905	93
12	4213597	?	23858	$(12, 6)$	462	3306	80

7 Tiling as an exact cover problem

We define a method *tilings* which takes as input a list L of polyominos, a polyomino P, and booleans *use_all_once*, *rotations*, and *reflections*, and outputs all tilings of P by elements of L respecting the boolean conditions on how we may manipulate the tiles.

tilings solves an exact cover problem (X, C) with DL. If we do not wish to use every tile exactly once, X is simply the squares of P. To obtain C: for each tile of L, for each permitted orientation Q of the tile, fix some square s_0 of Q; for each square s of P, translate Q such that s_0 coincides with s to obtain Q'. If Q' fits in P, add its vertex set to C. The permitted orientations of a tile are specified by the booleans *rotations* and *reflections*.

For list L in which every element is of size $|Q|$, the complexity of this initialization step is $O(|L||Q||P|^2)$.

If we wish to use every tile exactly once, we enumerate the elements of $|L|$ and add $|L|$ extra integers $t_1, \ldots, t_{|L|}$ to X; in the above procedure, if a translation Q' of the tile T_k fits in P, we add to C the union of the vertex set of Q' with the singleton $\{t_k\}$. This corresponds to adding $|L|$ columns to $M(X, C)$.

8 Polyomino tilings: some results

- Number of tilings of the given figures using all 12 free pentaminos exactly once, allowing rotation and flipping of the tiles: triangle - 374, diamond- 0, "stairs" - 404. (e.g Fig 5)
- Number of tilings of $n \times n$ square by fixed polyominos of area n: $n = 4 : 117, n = 5 : 4006, n = 6 : 451206$
- No rectangle tiled by all fixed/free/onesided 4-polyominos (we checked axb rectangles with $ab = 4P_{4,\mathcal{T}}$),
- There are 4040 tilings of a $5x12$ rectangle using all free 5-polyominos (e.g Fig 6)
- There are exactly 10 free octominos tiling their 4-dilate (Fig 8). Example of such a tiling: Fig 7.

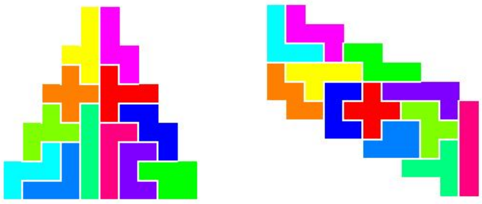

Fig 5

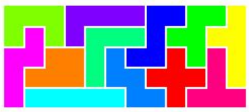

Fig 6

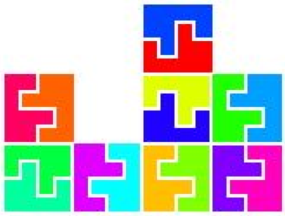

Fig 7

Fig 8

9 Sudoku solver

A sudoku problem asks us to fill a $9x9$ grid such that the numbers $1, \ldots, 9$ appear exactly once in every row, column, and in each of the 9 $3x3$ subgrids that compose the grid. Typically a number of squares are already filled in for us. Sudoku can be treated as an exact cover problem. There are four constraints on filling the grid which can be reformulated as sets to be covered:

- **Constraint A**: Every square must be filled. We need to cover the set X_A consisting of the 81 grid squares.

- **Constraint B**: Every row must contain the numbers 1-9. We must cover the set 81-element set X_B consisting of the pairs (r, i) of rows r and numbers $i \in [9]$.

- **Constraint C**: Similarly for the columns, we must cover X_C consisting of pairs (c, i) in $[9]^2$.

- **Constraint D**: Each of the nine $3x3$ subgrids must contain 1-9. We must cover X_D consisting of pairs (sg, i) where sg is the index of a subgrid.

We define the ground set $X = \bigcup X_i$. The exact-cover matrix $M(X, C)$ will thus contain $4 \times 81 = 324$ columns, with X_A corresponding to the first 81 columns, X_B the next 81, etc. Enumerating the squares of the entire grid from 1 to 81, we construct the set C as follows: For a square s of the grid which is not already filled in, for each $i \in [9]$ we consider placing i in s. This corresponds to covering one element of each X_i; we put these four elements in the a set $C_{(s,i)}$. If the square s was already filled in for us by some number i_0, we create just the set $C_{(s,i_0)}$. C is then defined as the union over all unfilled squares s of the $C_{(s,i)}$, and already-filled-in squares s of the $C_{(s,i_0)}$. $M(X,C)$ then has $9(81-c)+c$ rows where c is the number of clues given to us.

To define the matrix $M(X,C)$ we iterate through the squares (i,j) of the sudoku grid, checking whether they are already filled and, for each possible n, we place a 1 in the four columns listed in $C_{(s,n)}$. We store for each row of $M(X,C)$ the corresponding triplet (i,j,n) so that we can solve $M(X,C)$ by dancing links and look up what numbers to put in which squares.

Fig 9 Sudoku problem (23 clues) and one of its 96 solutions

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

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