Bulletin of the *Transilvania* University of Braşov • Vol 8(57), No. 2 - 2015
Series III: Mathematics, Informatics, Physics, 125-130

ALGORITHM FOR SOLVING A PUZZLE PROBLEM

Adrian DEACONU[1]

Abstract

We present an algorithm for solving a puzzle which consists in a set of p pieces that have to cover an m x n rectangle.

2000 *Mathematics Subject Classification:* 91A24.
Key words: puzzle game, backtracking.

1 Introduction

In this paper, we present an algorithm for solving the following puzzle problem:

- A rectangle of m x n squares it is given, where m and n are positive integer numbers.
- The rectangle is divided into p disjoint pieces, each piece is a combination of s_i squares (i = 1, 2, ..., p) so that:

$$\sum_{i=1}^{p} s_i = m \cdot n \tag{1}$$

- The pieces are scrambled.
- Solving the problem consists in placing the pieces on the m x n rectangle so that the pieces do not overlap and all the squares are covered.

There are applications based on this problem that can be downloaded from Google Play (Android), Windows Marketplace and App Market (iOS). Sometimes it is very difficult to solve the problem in the advanced levels and the implementation of the algorithm we propose can be a very useful tool to those who are not able to pass these levels.

Lets consider an example with m = 6, n = 5 and p = 5:

[1]Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: a.deaconu@unitbv.ro

```
oo
o
ooo

ooo
  o
  o
  o

 o
 o
 o
ooo
  o

o
o
o
o
o

oo
 oo
 oo
```

The solution is the following:

```
55222
45532
45532
41132
41333
41113
```

2 Preliminaries

The input of the problem can be considered as follows:

- A 0-initialized m x n matrix denoted R (the given empty rectangle)
and
- A vector of 0-1 matrices P_i (i = 1, 2, ..., p), the representation of the given p pieces (1 where square exists and 0 where square does not exist). On each margin (the first and the last line, the first and the last column) there must be at least one 1.

We denote by h_i and l_i the dimensions of the matrix P^i.
For instance, in the above example P^1 is:

110
100
111

We denote:

$$N\left(P^i\right) = \sum_{j=1}^{h_i} \sum_{k=1}^{l_i} P_{j,k}^i \tag{2}$$

We have:

$$\sum_{i=1}^{p} N\left(P^i\right) = m \cdot n \tag{3}$$

since the pieces cover the rectangle and they do not overlap.

This is a checksum test for the correctness of the input. If the checksum test is not passed it means that the input is incorrect, but if it is passed it does not necessarily mean that the input is correct.

The output is the pairs of coordinates (a_i, b_i) inside R for each piece P_i, for each $i = 1, 2, ..., p$, and:

$$1 \leq a_i \leq m \tag{4}$$

and

$$1 \leq b_i \leq n \tag{5}$$

so that the pieces cover the rectangle and they do not overlap.
For instance, in the above example we have:

$a_1 = 4, b_1 = 2$
$a_2 = 1, b_2 = 3$
$a_3 = 2, b_3 = 3$
$a_4 = 2, b_4 = 1$
$a_5 = 1, b_5 = 1$

3 The algorithm

We proceed with positioning the pieces inside R only if they pass the checksum test (see 3).

Well apply a backtracking strategy to solve our problem.

We try to position sequentially the pieces on the rectangle. We try every possible positioning of each piece till we place all the pieces. The k-th piece is

positioned at the coordinates i and j inside R if it fits inside R and it does not overlap with the previous $k-1$ placed pieces. The number of positioning of the k-th piece decreases with increasing of k because the number of empty spaces decreases and because the shape of the k-th piece matches more difficult with the previous placed ones. It is more efficient to position the pieces P with bigger $N(P)$. So, we can start the algorithm by sorting descending by $N(\cdot)$ in order to obtain a more time efficient implementation.

The pseudo-code of the recursive positioning is as follows:

```
bool SolveRecursively(k)
    if k = p+1 then
        return true;
    endif;
    for i=1 to m do
        for j=1 to n do
            if PieceCanBePositionedAt(k, i, j) then
                a[k] = i;
                b[k] = j;
                if SolveRecursively(k+1) then
                    return true;
                endif;
            endif;
        endfor;
    endfor;
return false;
```

The complete algorithm is the following:

```
Sort (P[i])i=1,2,...,n descending by N(·);
Sol = false;
if ChecksumTest(p, P) then
    if SolveRecursively(1) then
        Print(p, a, b);
        Sol = true;
    endif
endif
if not Sol then
    Print("No solution");
endif;
```

4 Conclusion and some possible extensions

We presented an algorithm to solve a puzzle which consists in a set of p pieces that have to cover an m x n rectangle. The pieces cannot be rotated.

A possible extension of the problem can be the situation when the pieces can be rotated. The implementation of this variant is more time consuming.

The input can be obtained by image processing using OpenCV by reading the pieces directly from a picture. This is also a possible extension of the application that makes it easier and interactive to use.

References

[1] D.J. Hoff, P.J. Olver, *Automatic Solution of Jigsaw Puzzles*, $http://math.umn.edu/\ olver/vi-/puzzles.pdf$

[2] P. Norvig, *Solving Every Sudoku Puzzle*, $http://norvig.com/sudoku.html$

[3] Sholomon, D., David, O. and Netanyahu, N. S., *A Genetic algorithm-based solver for very large Jigsaw puzzles*, $http://www.cvfoundation.org/openaccess/content-cvpr-2013/papers/Sholomon-A-Genetic-Algorithm-Based-2013-CVPR-paper.pdf$

[4] Wong, J., *The Fifteen Puzzle - The Algorithm*, $http://jamiewong.com/2011/10/16/fifteen-puzzle-algorithm.$