Tiling Puzzle Solver Project Report

Haoyu Chen (@virginia.edu)

Ke Dou (kd6ck@virginia.edu)

# Introduction

This report describes the algorithm and implementation of our Tiling Puzzle Solver Project, and it also includes some screen-shots of the program, showing snapshots of our program's execution. We evaluate the efficiency of our program, and give a table of CPU run times that our code required to find the solutions.

# Algorithm

Our algorithm is based on Dr. Donald Knuth's Dancing Links Algorithm, which can be found [here](https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/0011047.pdf). It solves an Exact Cover situation, so it can be extended to a variety of applications that need to fill constrains, such as this Tiling Puzzle Solver problem.

### Data Structure

We parse the input puzzle file into the Exact Cover matrix, and puts it into a doubly-linked list. For every column, there is a special ColumnNode, which contains that column’s Unique ID and the column’s size, the number of nodes in the column. Every 1 in the list, is a Node. Each Node points to another object up, down, left, right, and to its corresponding ColumnNode. A special ColumnNode h points to the first ColumnNode on the left as a starting point for the algorithm.

### The Algorithm

// if there is empty column, then it means in that case, there is no solution. return;

**if** (checkAllColumnEmpty(head)){

**return**;

}

**for**( Node rowNode = columnNode.getDown() ; rowNode != **null** ; rowNode = rowNode.getDown() ) {

solution.add( rowNode );

unCover( rowNode );

search( k+1, solution);

solution.remove( rowNode );

unCover( rowNode );

}

### Cover Function

cover( Node n ) function is the key of this algorithm. It removes a column from the matrix as well as remove all rows in the column from other columns they are in. we first remove the column from the other columns. Then we go down a column and remove the row by traversing the row to the right.

### Uncover Function

uncover( Node n ) is used to put the node back into the matrix. Since every node that has been removed retains information about its neighbors, we can use the reverse operation of cover to put the node back.

# Implementation

Processor: Intel(R) Core(TM) i7-3770 CPU 3.40 GHz

RAM: 8 GB

Programming Language: Java

JDK version: 1.7

IDE: Eclipse Luna Release (4.4.0)

GUI Toolkit: SWT

# Screenshots

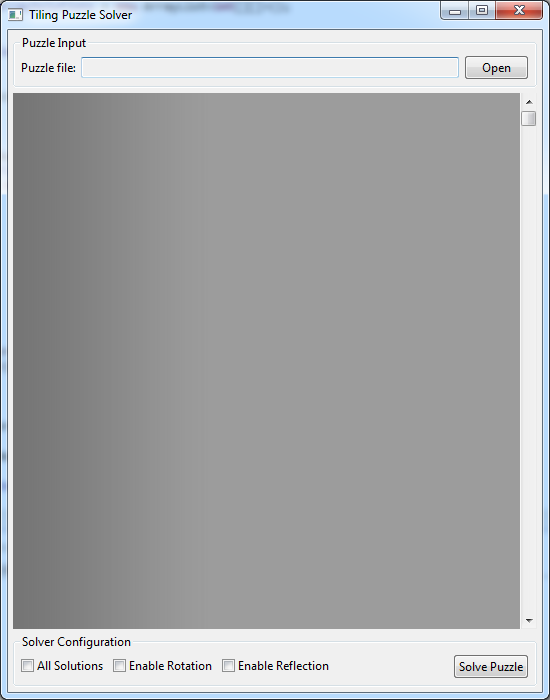


Figure: GUI after start up

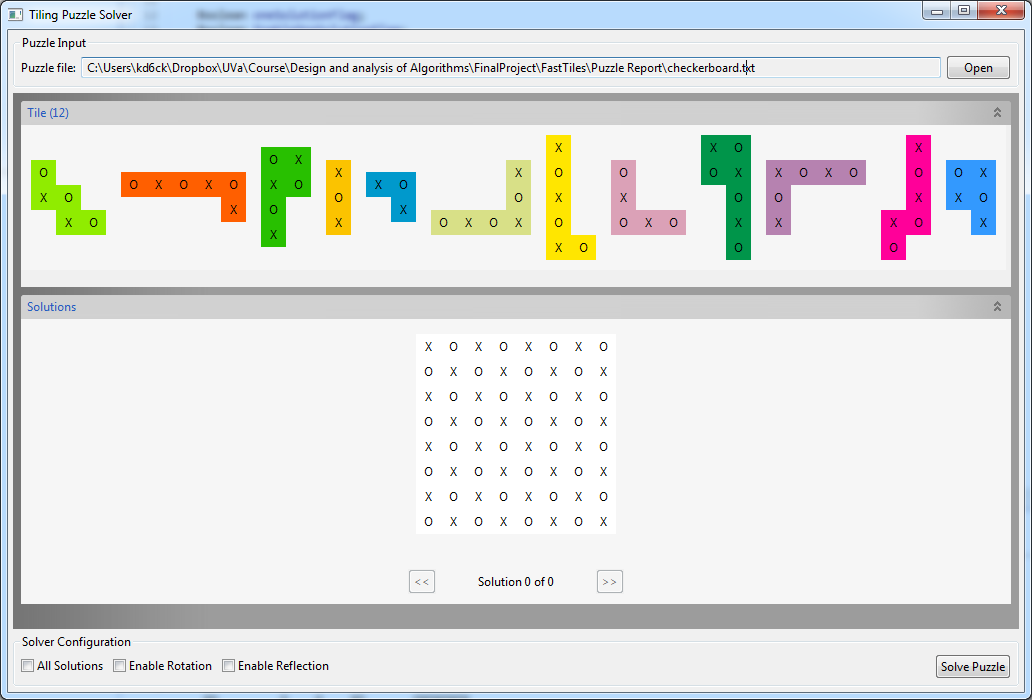


Figure: A puzzle is loaded from file

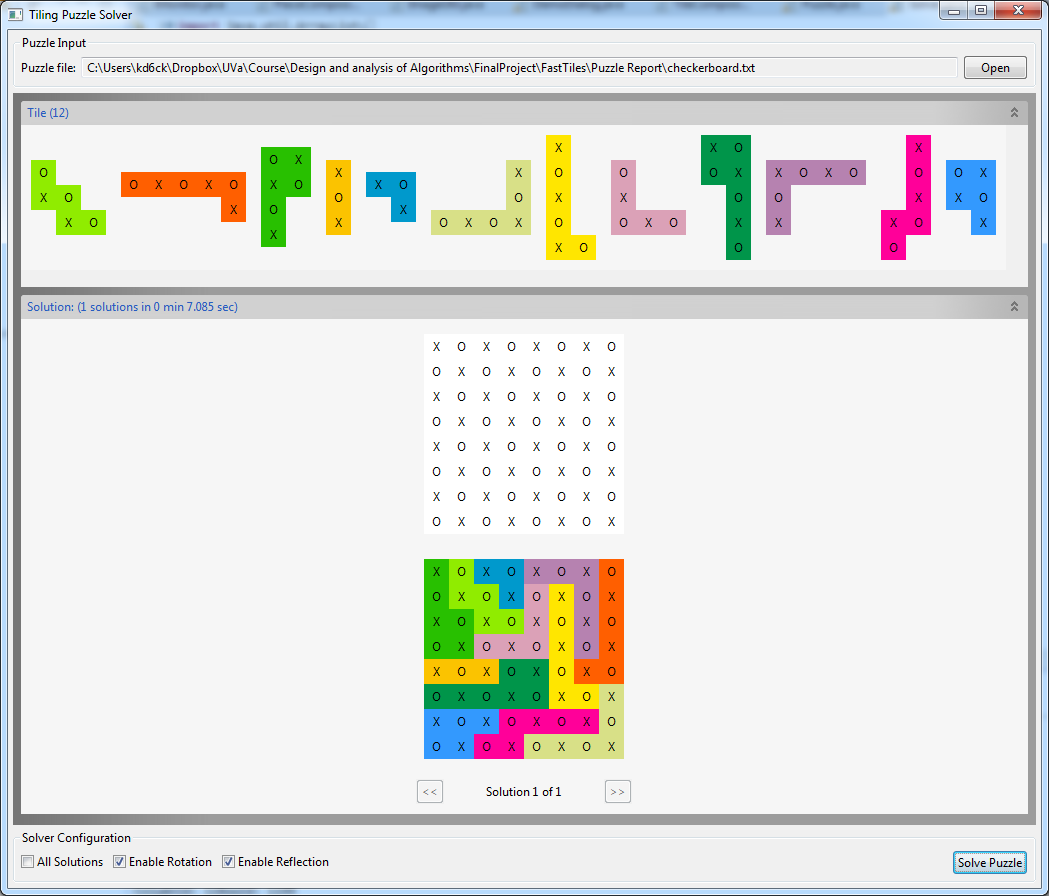


Figure: After puzzle solved

# CPU run times

|  |  |  |  |
| --- | --- | --- | --- |
| Puzzle name | Number of solutions | CPU time for 1 solution (sec) | CPU time for all solution (sec) |
| checkerboard | 6232 | 6.999 s | 73 min 9.991 s |
| IQ\_creator | 48 | 0.103 s | 4.265 s |
| partial\_cross\* | N/A | N/A | N/A |
| pentominoes3x20 | 8 | 9.185 s | 28.2 s |
| Pentominoes4x15 | 1472 | 1.668 s | 11 m 11.306 s |
| Pentominoes5x12 |  | 1.477 s |  |
| Pentominoes6x10 |  | 26.222 s |  |
| pentominoes8x8\_corner\_missing | 10054 | 1.765 s | 44 m 12.49 s |
| pentominoes8x8\_four\_missing\_corners | 1504 | 0.78 s | 6 m 16.12 s |
| pentominoes8x8\_four\_missing\_diagonal | 296 | 0.366 s | 1 m 40.46 s |
| pentominoes8x8\_four\_missing\_near\_corners | 1504 | 0.192 s | 6 m23.315 s |
| pentominoes8x8\_four\_missing\_near\_middle | 168 | 0.175 s | 1 m 38.313 s |
| pentominoes8x8\_four\_missing\_offset\_near\_corners | 216 | 0.279 s | 4 m 15.525 s |
| pentominoes8x8\_four\_missing\_offset\_near\_middle | 504 | 0.215 s | 2 m33162 s |
| pentominoes8x8\_middle\_missing | 520 | 1.577 s | 6 m 54729 s |
| pentominoes8x8\_side\_missing | 2576 | 5.431 s | 12 m 15.768 s |
| pentominoes8x8\_side\_offset\_missing | 1839 | 1.537 s | 10 m 47.539 s |
| lucky13 |  | 21.482 s |  |
| thirteen\_holes | 8 | 0.681 s | 1.586 s |
| trivial | 1 | 0 | 0.001 |

\*We don't handle the situation when the puzzle is partial-crossed because we didn't realize there is such a case until the last moment. However, the basic idea of addressing such case is to generate subsets of the participant tiles, and run our algorithm with each subset .