The University of Memphis
Department of Computer Science
Dunn Hall 375, Memphis, TN 38152-3240

Final Project
"Tiling Puzzle Solver"

Submitted By:
Nabin Maharjan
[U00534753]
Sujit Shrestha
[U00548000]

Due Date:
10th December 2014

Submitted Date:
8th December 2014

List of Figures

List of Tables

Contents

Introduction

A tiling puzzle consists of a solution board and number of input boards. Figures 1 – 5 form a puzzle. An input tile (or board) consists of a list of squares. A square has x-coordinate, y –coordinate and value. The objective is to tile the target board exactly by using the given tiles.

A board may be tiled in a number of ways using the input tiles. There may be many valid positions to place a tile. For example, Tile 1 may be placed at (0,0),(0,1),(0,2) and (0,4) positions in the board. Further, a tile may have multiple orientations. For example, Tile 4 has 8 different orientations; Tile 1 has two orientations while Tile 3 has only 1 default orientation.

So, a given puzzle can have many solutions. All these solutions may not be necessarily unique. A solution may be simply a reflection or rotation of another solution. All solutions must be checked for isomorphism to determine total unique solutions for a given puzzle.

Figure 1: Tile 1

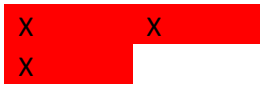

Figure 2: Tile 2

Figure 3: Tile 3

Figure 4: Tile 4

Figure 5: Board

Finding all solutions to a puzzle require exploring all possibilities of placing tiles to exactly cover the puzzle. The problem is of complexity O (2^n) and it may take a very long time depending upon the size of the problem and efficiency of solving approach. However, the time taken may be reduced if such problems can be processed in a distributed manner breaking them down into sub-problems and solving each sub-problem individually by different distributed processes.

The Tiling Puzzle problem is a type of exact cover problem and it can be efficiently solved using Knuth's Dancing Links algorithm (aka AlgorithmX). The algorithm models the problem as a sparse matrix represented by a circularly four-way linked list. The problem can be broken down into sub sparse matrices and hence be solved in a distributed manner.

In this project, we implemented Knuth's Dancing Links algorithm to solve tiling puzzle problems.

Approach

Solution Scheme

We modeled puzzle tiling problem as exact cover problem and then used Knuth's algorithm to solve the tile. So, each given puzzle is represented by a sparse matrix which is then solved using Knuth's algorithm.

The dimension of sparse matrix m x n is determined by the size of the puzzle.

m is the total number of rows and each row represents an input tile of any orientation that can be placed validly at a position of the solution board. Hence, creating rows filters out the tiles with invalid positions in the board.

n is the sum of the total number of input tiles and total valid squares in a solution board. Each valid square in the solution board is represented by a column and a separate column is created for each tile to represent each input tile. Therefore, each row in a sparse matrix has 1 at one of the columns representing the tile and 1 at the columns representing squares of the given tile.

For example, consider following puzzle with 4x 4 Board (in Figure 5) and 4 input tiles (Figure 1-4).

Here, total input tiles= 4, and total squares in the board: 4x4 = 16. So columns are created with labels (0-19). The first 4 columns 0-3 represent Tile 1 – 4 respectively, and the rest 4-19 column labels represent squares of the board in a left-to-right order. For example, the square at (0, 0) co-ordinate in the board has column label 4 while the last square at (3, 3) co-ordinate has column label 19.

Figure 6 shows a sparse row representing Tile 1(Figure1) at (0, 0) position of the Board (Figure 5).

1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0

Figure 6: A sparse row representing Tile 1 at (0,0) position of the board

Once the puzzle has been represented by its sparse matrix, the goal is to find a set of solution rows containing exactly one 1 in each column using Knuth's algorithm. The set of rows in case of puzzle problem will be any valid set of input tiles.

Knuth's Algorithm

The Knuth's Algorithm can be used to solve any exact cover column. As mentioned earlier, the tilling puzzle problem is also a case of exact cover problem.

We implemented the algorithm as described in Knuth's original paper: *Donald E. Knuth, "Dancing Links"*. The Knuth's Algorithm takes a sparse matrix as an input which is represented by a circularly four-way linked list structure. The problem is then solved using recursive backtracking algorithm (AlgorithmX) with cover and uncover operations and returns a set of valid rows that contains exactly one 1 in each column.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Figure 7: Input Sparse Matrix

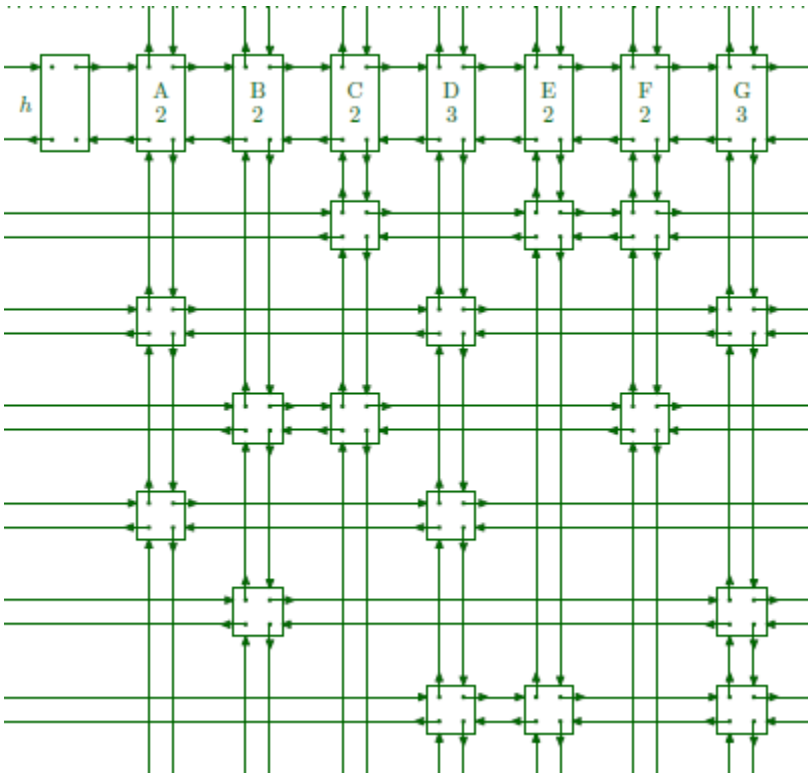

Figure 8: Four-way linked representation of sparse matrix in Figure 7

The cover and uncover operations form recursive backtracking for the algorithm. In case of covering column c, it removes c from the header list and removes all rows in c's from the other column lists they are in. Uncover operation in an undoing of the cover operation.

Search Strategy

We used greedy search strategy to find the column with the smallest number of 1's to cover at each step. All the rows in the covered column are then treated as a part of solution row set one at a time.

For the selected solution row, the columns of the 1s in it are covered. Then, the algorithm is called recursively to select another solution row. The solution set of rows are obtained if the sparse matrix is ultimately reduced to nil. Otherwise, the columns are recursively uncovered to backtrack as needed and then explore for remaining rows.

Optimization

The Knuth's algorithm is an efficient algorithm. In case of tiling puzzle problem, each sparse row represents a valid tile at a particular position in the board. So, the finally created matrix does not have any invalid tile representations.

Covering operation greatly reduces the size of sparse matrix at each step. Also, these operations are done in-place without any additional creation or deletion of objects by simply disconnecting and connecting links between the nodes and hence, these operations are very fast.

To further optimize, we selected the column in a greedy way; the column with the smallest number of 1s is selected for the cover operation at each level. This minimized the number of branching at each step.

Project Summary

The tiling puzzle solver project is a java project that solves the given puzzle in a distributed manner. The project consists of the following packages:

1. edu.memphis.clientServer

   The package consists of java classes required to implement client-server application model for solving the puzzle problem in a distributed manner.

2. edu.memphis.enums

   The package consists of java enum classes such as Orientation and OrientationMode.

3. edu.memphis.parser

   The package consists of java class to parse the given input puzzle file to obtain raw tiles and board.

4. edu.memphis.puzzle

   The package consists of java classes related with the representation of the puzzle. It includes classes such as Square, Tile, Board and Puzzle.

5. edu.memphis.serializable

   The package consists of serializable classes which will be used to communicate between client and server java processes.

6. edu.memphis.tilingSolver

   The package consists of java classes required to implement Knuth's algorithm.

7. edu.memphis.util

   The package consists of utility java classes.

How does application work

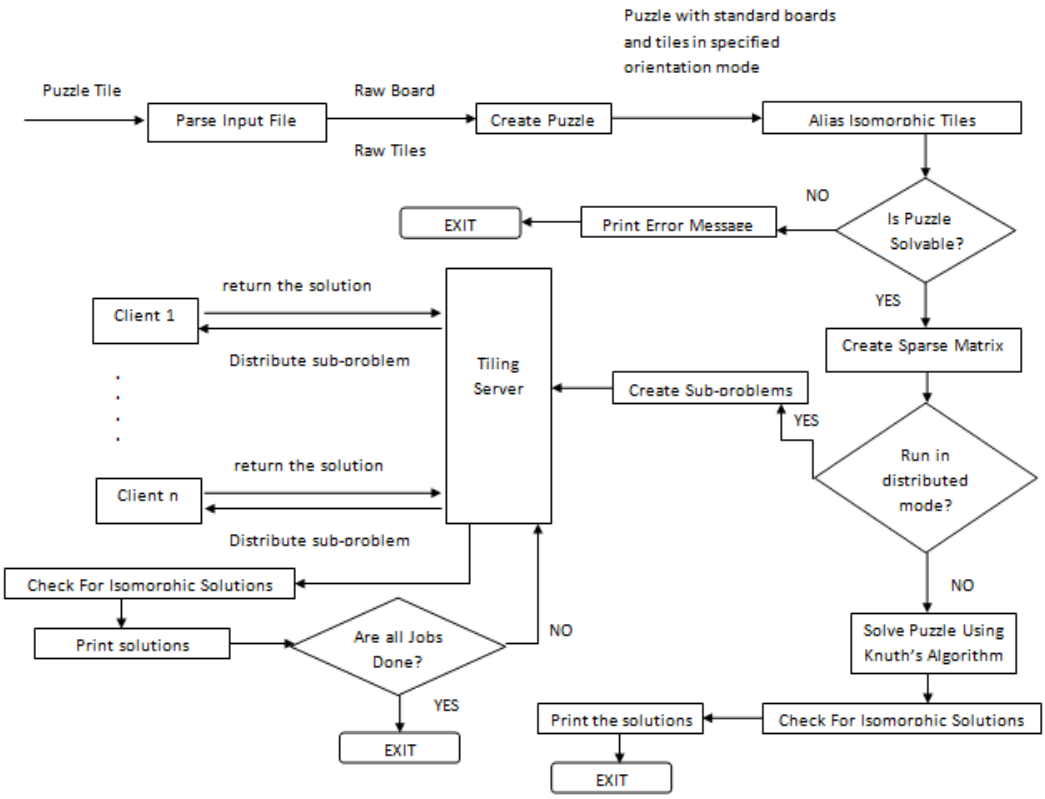

Figure 9: Program Flow for Tiling Puzzle Solver

The application is to be run from the command line and takes 6 arguments: *puzzleFileName*, *distributedMode*, orientationMode, *exploreLevel*, *numOfClientInstances* and *portNo*.

The *puzzleFileName* is the valid path of the puzzle file. The *distributedMode* argument can have any of {*true, false}* values. To run the application in distributed mode, it should be set to true. *OrientationMode* can be any of *{default, all, reflection, rotation}*. The tiles with be generated with the specified orientation mode. For example, in *all* orientation mode, all 8 possible orientations of each tile will be generated.

These first three arguments are mandatory and the last three requirements are not required if running the application in single instance mode. They should be specified to run the program in distributed mode. In distributed mode, if last three arguments are not provided, the application uses default values: *exploreLevel* = 1, *numOfClientInstances* = 2 and *portNo* = 3000.

The *exploreLevel* is used by the application to solve the original puzzle matrix to the specified level and then generates list of sub sparse matrices to be solved client working instances. Specifying higher value for *exploreLevel* results in larger number of sub-problems which is always better while solving problem in a distributed manner. However, higher value for *exploreLevel* may increase the cost of creating sub-problems before distributing to the clients. So, the optimum value should be selected for this argument depending upon the given puzzle.

Similarly, the number of clients to use can be specified for *numOfClientInstances* and the value for *portNo* argument can be given to start the server in that port for it to listen for the connection from the clients.

The application parses the input puzzle file and extracts raw board and raw tiles. These are then processed to create a puzzle object containing standard board and started input tiles. A standard tile (or standard board) has XY co-ordinates of its squares computed with respect to origin. The orientations of each standard tile are then computed depending upon user specified value for the OrientationMode. For example, all possible 8 orientations will be computed for each tile if the OrientationMode provided value is All.

Now, the default standard input tiles are checked whether they are isomorphic (same tile with same/different orientation). If they are isomorphic tiles, then they are given same alias id which will be later used while checking for isomorphic solutions to determine unique solutions.

At this point, the program checks for whether the puzzle is solvable or not. Since the puzzle problem is modeled as exact cover problem, the sum total of number of squares of all input tiles should be equal to the total number of squares in the board.

If the puzzle is solvable, the program creates sparse matrix representation of the puzzle. If running in single instance mode, the application solves the puzzle using Knuth's algorithm, checks for isomorphic solution and then finally, prints the total number of solutions and unique solutions.

In case of distributed mode, the application solves original sparse matrix to specified *exploreLevel* and then generates a list of sub sparse matrices. The application now instantiates Tiling server with a list of sub sparse matrices at a port value indicated by *portNo*.

The server accepts connections from the client instances. The total number of client connections accepted is determined by value for the *numOfClientInstances.*

The server assigns a client handler thread for each of the connected clients. Each client handler distributes a job (sub-problem) to its client and each client works on the sub-problem assigned to it and sends back solution to the client handler. The client handler in turn sends the solution to the server and checks if the server has any remaining job in its job list. If so, it removes a job from the list and sends it to the client for processing. If no job is remaining, it sends "no job remaining signal" to the client and terminates. The associated client exits upon receiving the signal.

The server checks for isomorphic solutions to determine unique solutions and then prints them. When all the client handler threads are terminated, the server exits.

Results

Below are tables showing the performances of the program for different number of client instances in the **memviz** cluster. The Table 1 lists the standard input puzzles while the Table 2 lists custom created puzzles. The table 3 and 5 shows the performance evaluation of the application given set of standard input puzzles and custom test puzzles created by us.

For each puzzle, the total number of solutions, number of unique solutions, CPU time for 1st solution and CPU time for all solutions are reported given different number of client instances 25, 50 and 100. The bold value is the best CPU time of the given puzzle in 1st Solution and All Solution categories when comparing among different number of client instances.

The puzzle board1 got killed for some unknown reason in the cluster before it could complete. One incomplete run reported 63493 unique solutions. So, Table 3 lists time to generate 1st solution only for board1. Please see Table 4 for puzzles 37 and 38.

Looking at the tables, it was found that application generally reported best CPU time for 1st Solution when the number of client instances was 25 and similarly, the best CPU time for all possible solutions was obtained when the number of client instances was 100.

Table 1: Standard Input Puzzles

ID	Puzzle Name	ID	Puzzle Name
1	board1	**20**	pentominoes8x8_corner_missing
2	board2	**21**	pentominoes8x8_four_missing_corners
3	board3	**22**	pentominoes8x8_four_missing_diagonal
4	board4	**23**	pentominoes8x8_four_missing_near_corners
5	checkerboard	**24**	pentominoes8x8_four_missing_near_middle
6	IQ_creator	**25**	pentominoes8x8_four_missing_offset_near_corners
7	lucky13	**26**	pentominoes8x8_four_missing_offset_near_middle
8	partial_cross	**27**	pentominoes8x8_middle_missing
9	partial_cross2	**28**	pentominoes8x8_side_missing
10	pentominoes_cross	**29**	pentominoes8x8_side_offset_missing
11	pentominoes3x20	**30**	pentominoes8x9_missing_middle
12	pentominoes3x20_shift	**31**	test1
13	pentominoes3x21	**32**	test2
14	pentominoes4x15	**33**	testBoard.txt
15	pentominoes4x15_shift	**34**	tetrominoes_shape
16	pentominoes5x12	**35**	thirteen_holes
17	pentominoes5x12_shift	**36**	Trivial
18	pentominoes6x10	**37**	hexominoes_knight
19	pentominoes6x10_shift	**38**	hexominoes_rook

Table 2: Custom created Input Puzzles

ID	Puzzle Name
T1	testPuzzle1
T2	testPuzzle2
T3	testPuzzle3
T4	testPuzzle4
T5	testPuzzle5

Table 3: Performance Evaluation against Standard Input Puzzles

		Client Instances	25						50						100					
ID	Number of Solutions	Number of Unique solutions	CPU time for 1st Solution			CPU time for All Solution			CPU time for 1st Solution			CPU time for All Solution			CPU time for 1st Solution			CPU time for All Solution		
			Hr	Min	Sec	Hr	Min	Sec	Hr	Min	Sec	Hr	Min	Sec	Hr	Min	Sec	Hr	Min	Sec
1*					**0.96**						1.58						2.1			
2	633856	18816			**0.8**	2	54	28.69			0.81	**2**	**6**	**54.93**			1.6	2	12	33.73
3	1000	1			0.22			**0.46**			0.24			0.51			**0.2**			0.47
4	The puzzle is not solvable																			
5	761	6232			**0.38**			4.14			0.55			2.94			0.5			**2.52**
6	48	6			**0.31**			1.35			0.49			**1.1**			0.6			1.23
7	555320	69415			**0.5**						0.79						0.89	5	53	35.9
8	The puzzle is not solvable																			
9	32	3			**0.12**			0.23			0.13			0.21			0.2			0.19
10	28	14			0.73			2.97			**0.69**			**2.83**			0.8			2.86
11	8	2			**1.79**			6.48			2.05			4.42			1.9			**3.07**
12	4	2			**0.58**			**0.73**			0.76			0.79			0.7			0.78
13	24	6			**0.38**			0.80			0.41			**0.57**			0.4			0.60
14	1472	368			0.6			8.74			**0.57**			5.38			0.9			**5.24**
15	276	138			0.59			2.26			0.63			1.74			**0.5**			**1.68**
16	4040	1010			0.6			21.72			0.56			12.92			**0.5**			**9.43**
17	466	233			0.5			3.34			0.66			2.59			**0.5**			**2.24**
18	9356	2339			**0.51**			34.97			0.64			21.43			0.8			**14.28**
19	312	156			**0.53**			3.83			0.60			2.75			0.6			**2.68**
20	10054	5027			**0.44**			35.61			0.60			29.17			0.7			**26.20**
21	17360	2170			**0.65**			47.56			0.69			29.98			0.7			**22.14**
22	296	74			**0.44**			3.61			0.54			2.41			0.5			**2.00**
23	1504	188			**0.37**			2.66			0.63			**2.16**			0.7			2.29
24	168	21			**0.38**			2.21			0.43			1.75			0.6			**1.56**
25	216	54			0.74			8.19			0.85			5.23			**0.7**			**4.49**
26	504	126			**0.34**			2.07			0.47			1.94			0.5			**1.80**
27	520	65			**0.35**			7.39			0.52			4.74			0.7			**3.85**
28	2576	1288			**0.53**			18.23			0.59			10.50			0.9			**7.71**
29	1839	1839			6.41			16.72			3.39			17.35			**2.1**			**9.99**
30	36	9			0.49			1.05			**0.44**			**0.71**			0.6			1.08
31	384	1			0.07			0.21			0.07			0.20			**0.1**			**0.20**
32	0	0			0			0.00			0.00			0.00			0.0			0.00
33	The puzzle is not solvable																			
34	8	1			**0.23**			0.39			0.3			0.3			0.27			**0.3**
35	8	2			**0.22**			0.32			0.3			0.3			0.23			**0.3**
36	1	1			0.07			0.07			**0.1**			**0.1**			0.06			0.1

Table 4: Puzzles taking too long time to solve

ID	Remark
37	The puzzle size is too big and also the program did not report first solution even when running for a long time with 100 instances. May be it has no solution? We did not run it completely due to time constraints.
38	Same remark as for puzzle ID 37

Table 5: Performance Evaluation against Custom Input Puzzles

		Client Instances	25						50						100					
SN	Number of Solutions	Number of Unique solutions	CPU time for 1st Solution			CPU time for All Solution			CPU time for 1st Solution			CPU time for All Solution			CPU time for 1st Solution			CPU time for All Solution		
			Hr	Min	Sec	Hr	Min	Sec	Hr	Min	Sec	Hr	Min	Sec	Hr	Min	Sec	Hr	Min	Sec
T1	50612	50612			0.53		24	34.68			**0.46**		23	58.1			0.76		**23**	**34.19**
T2	1134	1134			0.33			1.28			**0.32**			**1.1**			0.71			1.54
T3	16	2			**0.24**			**0.36**			0.35			0.4			0.36			0.40
T4	81	81			**0.24**			0.84			0.36			**0.7**			0.59			0.80
T5	74034	37017			21.53		**25**	**14.77**			7.32		25	57.0			**0.55**		25	46.92

References

1. *Donald E. Knuth, "Dancing Links"*.