Report of my project on INF421: Logipix

1- Choice of structure:

I first thought to use an object cell that has neighbors stored in his attributes and so on but finally I found the idea quite heavy and that this could let me use many classes and many dynamic objects so I decided to use *arrays int[2]* to represent cells and to look for their neighbors using private methods and a two matrixes this.puzzle and this.state to represent the logipix. (than the other attributes are used because I needed them in the coming questions)

2- Representation and printing:

I decided to represent the puzzle in a matrix (natural representation), states in another matrix so that 0 will be unmarked cell 1 maybe marked and 2 marked, printing a puzzle shows a matrix of values and states and drawing a puzzle shows a draw where zeros are represented with “X” and ones or twos with “ “.

Example of printing LogiX state after creation (all cells to state 0):

```
Puzzle Logix :
[ 2|0,  2|0,  3|0,  0|0,  0|0,  0|0,  0|0,  0|0,  0|0,  4|0,  1|0]
[ 0|0,  1|0,  0|0,  0|0,  0|0,  0|0,  0|0,  0|0,  0|0,  1|0,  0|0]
[ 0|0,  0|0,  3|0,  1|0,  0|0,  0|0,  0|0,  1|0,  4|0,  0|0,  0|0]
[ 0|0,  0|0,  0|0,  4|0,  0|0,  1|0,  0|0,  6|0,  0|0,  0|0,  0|0]
[ 0|0,  0|0,  0|0,  3|0,  0|0,  6|0,  0|0,  3|0,  0|0,  0|0,  0|0]
[ 0|0,  0|0,  0|0,  0|0,  4|0,  0|0,  0|0,  0|0,  0|0,  0|0,  0|0]
[ 0|0,  0|0,  1|0,  3|0,  0|0,  0|0,  0|0,  3|0,  3|0,  0|0,  0|0]
[ 0|0,  1|0,  4|0,  0|0,  0|0,  0|0,  0|0,  0|0,  0|0,  2|0,  0|0]
[ 4|0,  0|0,  0|0,  0|0,  0|0,  0|0,  0|0,  0|0,  3|0,  2|0,  1|0]
```

Example of a Logipix drawing:

```
A draw of the solution :
[ ,  ,  , X, X, X, X, X,  ,  ,  ]
[X,  ,  , X, X, X, X, X,  ,  , X]
[X, X,  ,  , X, X, X,  ,  , X, X]
[X, X, X,  ,  ,  ,  ,  , X, X, X]
[X, X, X,  ,  ,  ,  ,  , X, X, X]
[X, X, X,  ,  ,  ,  ,  , X, X, X]
[X, X,  ,  , X, X, X,  ,  , X, X]
[X,  ,  , X, X, X, X, X,  ,  , X]
[ ,  ,  , X, X, X, X, X,  ,  ,  ]
```

3- Broken lines of a clue:

Generating all the broken lines of a clue needs to go through cells and their neighbors respecting the state of the puzzle means marked or maybe marked cells are never considered in neighbors and to be efficient we need to stop whenever we reached a number of piles in the broken lines which is equal to the clue we started from is value.
This method does at max:
4 (four first neighbors) * 3 (3 neighbors every time for the cells that come after the clue) ^ (value of the clue – 1) = 4*3^(clueVal-1) = O(3^clueVal)
We cannot do better at this stage if a cell has an empty neighborhood we must do all the ***4*3^(clueVal-1)*** steps I mean like in :

0 0 0 0 0
0 0 0 0 0
0 0 3 0 0
0 0 0 0 0
0 0 0 0 0

For example, in 'test.txt' puzzle:

6 0 0 0 0 6
0 0 0 0 1 1
0 0 0 6 0 0
0 0 0 5 1 0
0 0 5 1 0 0
0 0 1 1 1 1

The set of broken lines of the clue (0,0) is:

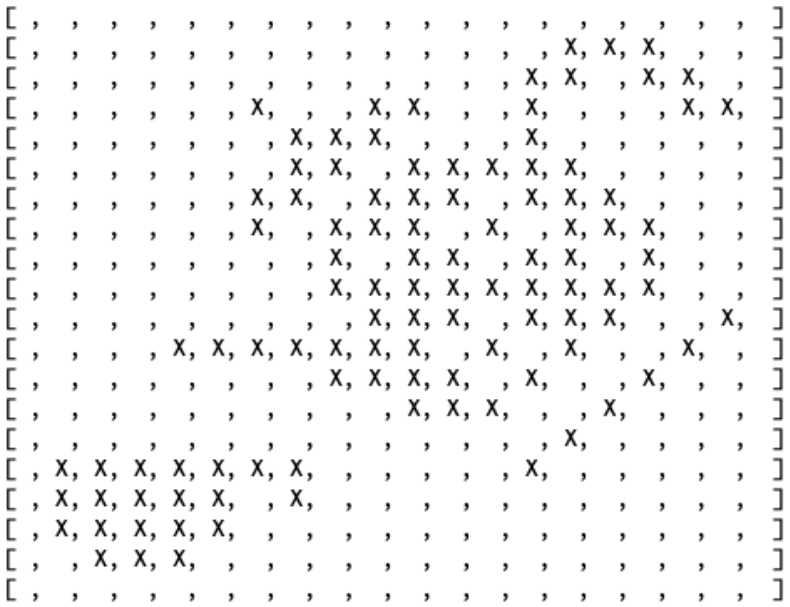
[(0,0)->(1,0)->(2,0)->(2,1)->(2,2)->(2,3)->]->[(0,0)->(1,0)->(1,1)->(2,1)->(2,2)->(2,3)->]->[(0,0)->(1,0)->(1,1)->(1,2)->(2,2)->(2,3)->]->[(0,0)->(1,0)->(1,1)->(1,2)->(1,3)->(2,3)->]->[(0,0)->(0,1)->(1,1)->(2,1)->(2,2)->(2,3)->]->[(0,0)->(0,1)->(1,1)->(1,2)->(2,2)->(2,3)->]->[(0,0)->(0,1)->(1,1)->(1,2)->(1,3)->(2,3)->]->[(0,0)->(0,1)->(0,2)->(1,2)->(2,2)->(2,3)->]->[(0,0)->(0,1)->(0,2)->(1,2)->(1,3)->(2,3)->]->[(0,0)->(0,1)->(0,2)->(0,3)->(1,3)->(2,3)->]->[(0,0)->(0,1)->(0,2)->(0,3)->(0,4)->(0,5)->]->

(Run test.java)

4- (see below).
5- First solving approach: (4th quest + optimizations)
Using a priority queue that puts clues in order of values with a bigger priority to clues having a smaller value in puzzle, we do an efficient backtracking since we start with things that if we backtrack we rapidly get to the right broken line (if we back track from 2 and go back to this 2 again we will be in the right broken line but if we start from 15 we do unnecessary stuff and if the 15th broken line is the right one we will do a lot of them).

The complexity of this method is in the worst case (I mean where always the last broken line is the right one in all clues) is something like ***O(3^sum(clues values))*** !! which is quite a lot.

This solved provides the solution of Teacup for example in: 44s

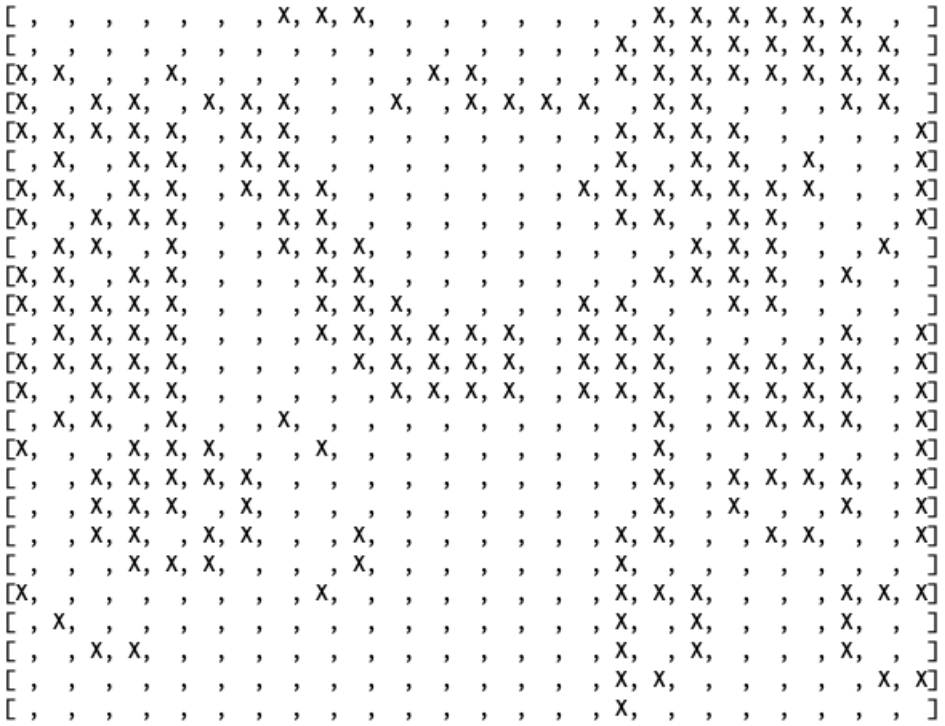


6- Combinations and exclusions:

Here we implement a natural idea that humans use to solve such puzzles (also used and more clearly in sudokus) the idea of exclusion it gets us to a lot of loops but still better that the tremendous complexity of the solver above, it has a complexity of ***O(#clues^3 * height * width * 3^(max clues))***.

The idea of the algorithm chosen here is the natural idea of starting with the cells used by all broken lines in a clue than looking whether some clues still have only one broken line left than update all clues broken lines given this information and repeating this until we got the maximum from exclusions, this gives already quite a good shape of the solution when drawing the output result.

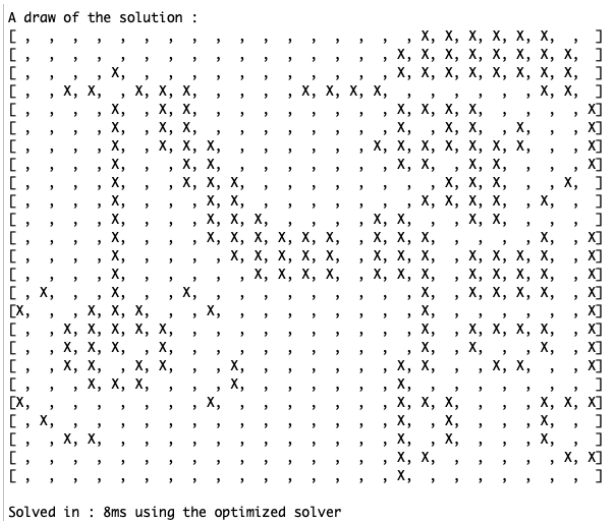
For example, after applying this on Man puzzle, we got this:



7- Second solving approach (using quest 6)

we just need to do the same of what we have done in 5 using only useful broken lines : note that one on the problems that took me a lot of times to debug is that after each step of the backtrack I forgot to take at account the state of the puzzle to eliminate marked broken lines from the useful ones (this is why I used the method ***setOfUnmarked()***) It worked quite well and solved puzzles quite rapidly in comparison with 5 : solved Teacup in 440ms and Man in 8ms!

Here is a preview of the puzzle Man solved using this method:



8- Optimizations

All what I thought of as optimizations are used in the above methods among them : I tried to save the most part of the work so that time complexity improve (memory is

not a big deal in such small puzzles), I also tried to not recalculate broken lines for already seen clues having one broken line in the while loop of quest 6 and I used injections to code tuples with Integers so that we do not use heavy objects as keys in addition to the problem that ***(new int[]{1,2} != new int[]{1,2})*** which complicates the task of using arrays as keys of a HashMap..

⇨ Please check the top of the file logipix.java for more details on the *algorithmic* part of the questions.