Algorithms

Validator

The Validator algorithm has the objective of Validating if a Solved Hidato is valid or not. It has an inner class called Coords that has X and Y as attributes. The way the Validator Algorithm works is quite simple.

First of all, searches for the position of the number 1 on the matrix. Once it gets it, creates a Coord with the position of it. Next to it finds the maximum number of the matrix and stores it's value. Finally, with a variable called num = 1, while this num is lower than the max number, using the getNeighbours method searches for all the neighbours of the first number. Once it has all his neighbours, searches if there is any neighbour that values 2. If it does not, just ends the process and returns false. If the 2 exists, it updates the coords of num with the next one, and repeats the process searching for all the neighbours of number 2 and searching for the next number to be his neighbour. If the while loop gets to and end without return false it means that the Hidato is well done so it returns true.

Generator

The generator algorithm has the objective to generate a Matrix with a template of a solvable Hidato. It has an inner class called Coords that has X and Y as attributes.

The generator starts by creating an instance of a problem depending on the type demanded by the player. The attributes of the problem instance are given through parameters. Then creates a matrix of strings, at first this matrix is filled with “?”, this means that in this position there could possibly be a number. After this we fill in some “#” making sure that the map is still solvable at this point. “#” represent a border from the map (it makes sure that there are no locked “?”). Once the template is done the next step is filling in the map with a solvable hidato. Depending on the size of the map, the type and the adjacency our algorithm is capable of finding solvable Hidatos that fill a minimum percentage of the board. It starts by a random position in the map, then a recursive function fills in the given position from the array with a number found in the parameter of the function (the first number is always 1), calls a function called getNeighbours that returns all the available neighbours of the position the function has already filled and shuffles the given array in order not to access always the same neighbour and then calls itself recursively. Once there are no more available neighbours the function ends returning the biggest number filled in

the map. If this number is bigger than the percentage calculated before calling the function the Hidato is valid, else the process starts again until it satisfies the condition.
Once the Hidato is found in the map the user can request a difficulty. There are 6 available difficulties, the higher the difficulty is the less numbers are lef in the map. Currently the percentages of empty cases from the easier to the harder are: 20%, 35%, 50%, 65%, 80%, 90%. Through a deep copy function (function that copies one by one each position from the matrix) the algorithm takes randomly numbers that are or 1 or the maximum, satisfying again the percentage..

Finally the Hidato is returned and stored if the player wants to.

Solver

The solver algorithm has the objective of solving an Hidato game alone, without been helped by a player. This algorithm is in the Solver.class and it has an inner class called Tuple which has the attributes X, Y and “agafar”. The first two are int variables that we will use to move around the matrixes.The last one is a boolean which will tell us if the position is a good Neighbour for our actual matrix position.

This algorithm works with the Backtracking technique. It generates all the possible solutions for that Hidato game. But, first of all, it searches for the position of the number 1 in the matrix. Once it gets it, the algorithm will look for all possible Neighbours of the actual position, using the functions NeighboursS/T/H, being S for Square, T for Triangle and H for Hexagon; like a sonar. After that, the function goodNeigbours() is called and it selects only the neighbours that are considered good by making true the boolean “agafar”. This function can return 2 possible arrays. If one of the neighbours is the next number of the sequence, returns an array with only one position; altarwise, returns an array that has between 1 and the maximum number of neighbours.

Finally, the algorithm will search for all the correct positions to complete the game with the recursive call of the function Backtracking. Solve will print the first solution it finds and descart other options to make it a little more efficient.