JAVA Advanced Project: Infinity Loops

——M1——

1 Introduction

The aim of this project is to work around the game Infinity Loop (see Figure 1). In this game, a level consists of a grid where each square(cell) can have (or not) a piece inside. The goal of the game is to make all the pieces connected. In this project we assume each cell may be empty or it can take one of the pieces represented in Figure (2).

A cell can be seen from 1 to 4 of its neighbors. A cell can be rotated by 90 degrees. An example of rotation is given in Figure 3. Note that the number of rotations are not the same for all the pieces, there might be a piece with four possible rotations, three possible rotations or even zero rotations. A level is solved if all the components are connected.

The goal is to have:

- A level generator.
- A solution checker.
- A level solver.
- A visualization of a level.

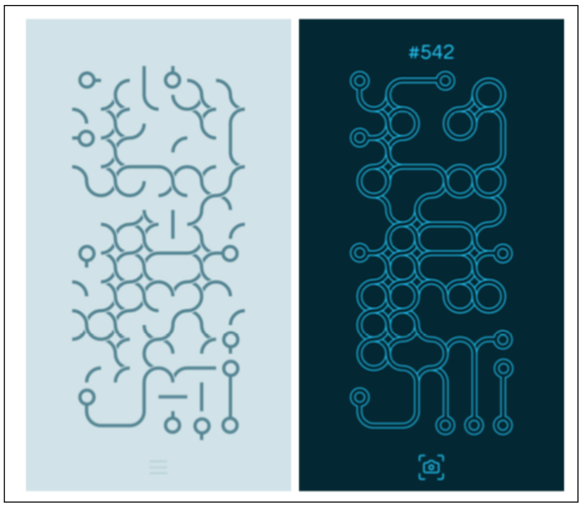

Figure 1: The left image: a level of infinity game with two connected components. The right image: a solution to the level.

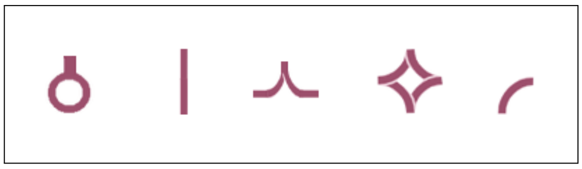

Figure 2: The possible pieces in each cell.

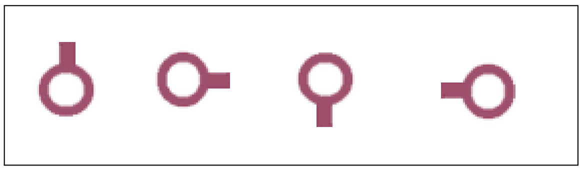

Figure 3: For this piece four rotations are possible.

2 To be done

2.1 A Level Generator

To generate a level of the game, the class Generator is provided to you. The goal of this class is to generate a grid based on the options specified in the command line. The command to run the generator will be :

$$java - jarprojet.jar - -generatewxh - -outputfile[- - nbccc]$$

Where file is the name of the output file, w and h the dimensions, and c is the maximum number of the connected components in the generated level (optional, the program must be able to run without $- -$ *nbcc*).

At the end of the execution, the generator must have created a text file ”file” containing a description of the grid. The file must follow the format of writing as in section 3.

To construct a level and mixing the initial positions you can follow the instructions below:

To construct a level, you can start from a corner of the grid and then determine the piece presented on each cell step by step. For each cell, choose randomly among the possible pieces (or the absence of a piece) by taking into account the borders of the screen and the neighbors already placed. The initial position of each piece is then mixed. It is specified that a level can several ”connected components” as in Figure 1. Optionally, the generator can be asked for a level where the number of connected components of the solution is specified.

To write this class, you need to consider *width* and *height* of the grid as well as the number of connected components($- -$ *nbcc*) that are demanded. If the number -1 is specified for $- -$ *nbcc* it means there is no connected components and you can choose pieces and their orientations randomly (by considering the corners and borders of the grid). If the number of connected components are specified, you should look for a way to integrate it in the level generation process.

To register the grid is a file you can use *NIO2* package to write the result in the destination file.

2.2 Solution Checker

This program takes as input a file consisting of a potential solution and displays on the standard output "SOLVED: true" if it is a solution or "SOLVED: false" otherwise. The command to run the checker will be :

$$java\ -jar\ projet.jar\ --check\ file$$

where file is contains a grid to be checked, it is written in the format specified in section 3. What you need to do is to generate a grid based on the provided solution in the file and then to implement a method *isSolution*. The checker should check if all the pieces are connected or not. If all the pieces are connected it means that the game is solved.

2.3 Level Solver

This program takes as input a grid and attempts to solve it. The command to run the solver will be:

$$java\ -jar\ projet.jar\ --solve\ file\ --output\ filesolved\ [--threads\ t\ ]$$

where *file* is the file containing the grid to be solved and *filesolved* the solved grid. Both files must be written in the format specified in section 3 and contain nothing else. The solver must also display "SOLVED: true" on the standard output if the grid could be solved or "SOLVED: false" otherwise.

The optional argument must allow to fix the number of threads used for solving the problem. Be careful, if we call the solver with the argument -t 1, the solver must not use more than one thread. If the argument is not given, the solver must use only one thread.

The class Solver is for solving a grid. The main task to do for this class is to write the main method containing methods and strategies to solve a grid. In the next section, some strategies to solve the grid are proposed.

Solving method

Exhaustive Solving

A simple way to solve the problem is to test all the positions of each piece in all cells and check if it is a solution. We will assign positions to the pieces and test this assignment. Each piece can have several positions. In other words, we go through the search tree of the positions of the pieces, where the leaves correspond to a complete assignment of all the pieces. Given a partial assignment (when only a part of the assignment of the pieces is known), three possibilities exist:

- The conditions of the resolved level are met.
- It will not be possible to solve the level given the current choices: one must go back.
- The level is not yet solved, a new (as yet unstudied) piece is chosen and the game is rerun for each possible position of this piece.

Think about the problem as a recursive problem. For the purpose of efficiency, we will prefer to simulate the recursiveness with the help of a stack (we pile up the possibilities of the positions of the pieces). The possibilities are de-stacked and tested until an empty stack is obtained.

Choice of the piece to be tested

The choice of the piece to be tested has a great influence on the speed of the algorithm. One can consider a completely random choice, a choice according to the number of neighbors of the part, according to what is already fixed in the game, from left to right, according to what is affected if one fixes its position,... . Beware, a more sophisticated rule can speed up the general algorithm, yet if it takes too long to find the piece, you lose its effect. You must implement at least 2 different principles of piece selection, which can be chosen in a clean way at the code level. You can consider an optional argument on the command line (with a default value for the script to work), an automatic choice depending on the input formula, or at least a way to choose it by modifying only one line of the code. Be aware that there may be several different solutions for the same level.

2.3.1 Algorithmic improvements

It is not necessarily useful to test all pieces. For example, a piece that can meet its 4 neighbors does not have several positions. A piece at the edge of the board does not have all its positions valid. In addition, based on the current position of a piece it might be possible to eliminate some of neighbors' piece positions.

These checks can be done before and/or during solving process.

Improvements via multi-threading

Concurrent programming can be used to speed up the solving process, so the solver should take as input a parameter - T x where x is the maximum number of threads the program can run.

Several approaches are possible:

- Portfolio approach: Several processes run in parallel with different choices of variables (different choice algorithm or different seed of the random). The first one to find a solution tells the others to stop.

- Separation of the search space: if a certain part is chosen, one can imagine a process for each position of this part. For this purpose, a master/slave model can be considered where the master distributes tasks to the slaves who compute. The master maintains a list of tasks to be performed. When a slave is free, it asks the master for a task. Beware, this improvement is difficult to implement.

If an instance of a level is given that is not solvable, the solver must write SOLVED: false on the standard output. A more efficient method than testing all positions to detect this will have to be found, otherwise the resolution time will be too long.

2.4 Graphical Interface

For debugging, you can use a display on the terminal using unicode. You can find these unicodes here. Optionally, a graphical interface is required to visualize a level or a solution. In the classes I have used swing, but if it is more convenint for you do not hesitate to use JavaFX.

Level 0 of this GUI is simply to statically visualise a level (solved or not) in a nicer way than with unicode characters. To have this minimized display of result implement the method *buildTable* of the class DisplayUnicode.

A first evolution consists in giving the user the possibility to play (click to rotate the pieces). Another evolution consists in being able to visualise the solver working, i.e. to see the rotations of the pieces live and thus to see the state of the level by the solver. To do that implement the methods *initialize* and *getImageIcon* of the class GUI.

3 Integrating all parts of the program

As you can see, to communicate with different parts of the code, we use the command line. To this end, in the class *main* I have give you a parser to control the instructions. All you need to do is to call appropriate methods in their right place. Can you propose an alternative way to read command line arguments using the contents of the course? If yes, it would a **plus** to implement it.

Since each time the grid should be registered in a file, it is essential to have a specified format to register the grid. In this project, use the following format:

Format in its basic form, the game has 6 types of pieces:

0. Empty cell.
1. Those with only 1 neighbour.
2. Those with 2 opposite neighbours. (the bar)
3. Those seeing 3 neighbours (a T).
4. Those seeing 4 neighbours (a cross).
5. Those seeing 2 consecutive neighbours (an L)

Therefore the files should respect the following format.

- first line: width of the grid w
- second line: height of the grid h
- from the line 3 to the line $w.h$: the piece-number (i, j) and the piece orientation number (i, j).

We place the pieces in the file from the top-left piece of the grid, to the bottom-right of the grid : assume that the piece at the position $(0, 0)$ of the grid is at line 3, the piece at the position $(0, 1)$ is at line 4, and so on.

To have a concise representation of pieces and their orientations, use the information in the table in Figure 4 to implement the enumerates for *PieceType* and *Orientation*.

You will find some clues on how to write the appropriate enumerations by inspecting the classes Grid and GUI.