INF421 PROGRAMMING PROJECT
SOLVER OF LOGIPIX

MARIE ALBENQUE
albenque@lix.polytechnique.fr

The required programming language for this project is Java. All the code you submit must be commented and indented properly and must be organized logically in some classes.

The purpose of this programming project is to write an efficient solver for *logipix puzzles*, such as the one below. The goal of a logipix puzzle is to discover an hidden picture thanks to some numerical clues. Each puzzle consists of a grid containing easy empty cells or cells with numbers. Every number, except for the 1's, is half of a pair. The object is to link the pairs and painting the paths so that the number of squares in the path, including the squares at the ends, equals the value of the clues being linked together. Squares containing 1 represent paths that are 1-square long. Paths may follow horizontal or vertical directions, can turn and are not allowed to cross other paths. Such a path will be called a *broken line* in the rest of the text.

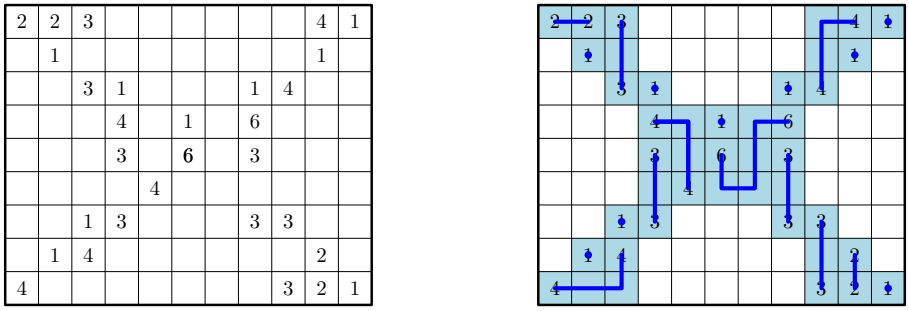

FIGURE 1. A logipix puzzle (left) and its unique solution (right).

This programming project is of increasing difficulty. The first part consists in creating classes that can handle logipix puzzles, in particular instantiate and display them. The second part consists in writing a brute force algorithm that solves a logipix puzzle. The third part improves the efficiency of the solver.

Before typing some codes, take a moment to think about the organization of your project into classes. Consider the different options for the needed data structures and pick the most-appropriate ones, justify your choices in your report.

1. Setting up the game

1.1. **Reading a file and instanciate a game.** A logipix puzzle will be given by a file of the following format. The first two lines give respectively the width and the height of the grid. Then each line gives the clues line by line, where the empty cells are represented by zeroes.

For instance, the first lines of the file encoding the game given in Figure 1 are:

```
11
9
2 2 3 0 0 0 0 0 0 4 1
0 1 0 0 0 0 0 0 0 1 0
0 0 3 1 0 0 0 1 4 0 0
0 0 0 4 0 1 0 6 0 0 0
...
```

and the full code is available here : www.lix.polytechnique.fr/~albenque/Logipix/Examples/LogiX.txt. Other input files are available in the directory www.lix.polytechnique.fr/~albenque/Logipix/Examples and can be used to test your code.

Task 1. *Write a method that takes such a file as input and initializes a logipix puzzle. (Don't forget to give some thoughts about the organization of your project into classes and about the data structures you want to use!)*

A *state* of the puzzle, is the initial configuration of clues together with some *partial solution*, i.e. the cases can be either colored, not-colored or "maybe colored".

Task 2. *Write a method that displays a state of the puzzle. This can be a very simple representation or a fancier one if you feel like it (but there won't be extra credit for fancy graphical interface).*

In the rest of the project, we are always interested in finding *one* solution of the puzzle.

2. Solving a logipix puzzle with backtracking

The first method we are going to use to solve a logipix puzzle is quite brutal. The idea is the following. For each clue, we generate the set of all possible broken lines. We start with the first clue and pick one of the possibility. Then we move to the second clue, do the same, while ensuring that this is compatible with the possibility we picked for the first clue and so on and so forth. If, at some point, we run out of possibilities, we change one of the choice we made before, hence the name backtracking. For instance for the grid below (which represents the top-left corner of one of the grid given in the examples), the first steps may be something like that (it depends on the order we choose between the different possibilities):

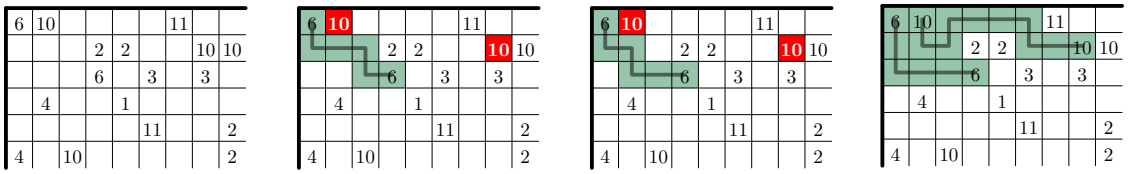

Figure 2. The beginning of a solver based on backtracking.

A *state* of the puzzle, is the initial configuration of clues together with some *partial solution*, i.e. some of the cases are declared to be "assumed-to-be-occupied" if they are part of the solution considered by the backtracking algorithm (i.e. those are the green cells of Figure 2).

Task 3. *Write an (efficient!) method that generates all the possible broken lines corresponding to a given clue and which are compatible with the current state of the grid.*

Task 4. *Design a first logipix solver using the backtracking approach described above using the method implemented in Task 3.*

Even if using backtracking is a brute-force method, try not to be too brutal ! In other words, to improve the complexity of your solver:

- Think of the ordering in which you consider clues. Note that if the current state of the grid presents few constraints, the number of possible broken lines associated to a given clue grows exponentially with the value of the clue.
- Try to backtrack as soon as possible by identifying some "dead-ends".

Task 5. *Improve your solver by using these ideas and compare the running times.*

When the sizes of the grid and of the clues increase, the backtracking-based solver becomes less and less efficient. We will see in next sections how to improve its efficiency.

3. Combination and exclusion

We now consider combinations of partial possible solutions. Generate all the possible broken lines corresponding to a given clue. If the intersection of all these lines, if it is non empty, then we can conclude that the cases in the intersection belong to the solution. We illustrate this idea in Figure 3. In a state of the puzzle, some cases can now be "occupied" (clues or blue cases) and will remain occupied until the end of execution of the solver.

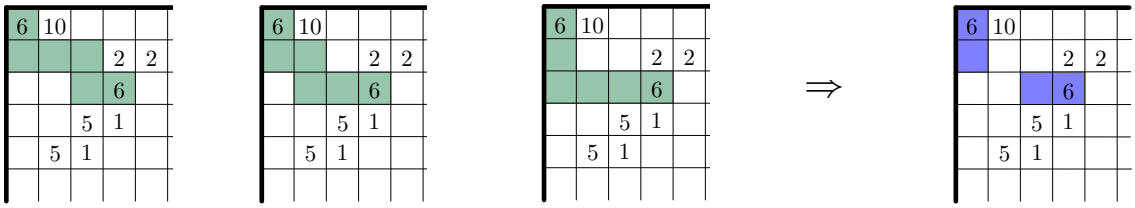

Figure 3. On the left, the 3 possible broken lines to link both '6', on the right, we identify some of the cases that are necessarily parts of the solution.

After identifying successfully some cases of the solution, some lines are not compatible anymore with the new current state. We can then apply "exclusion", remove these lines and possibly identify some new occupied cases, see Figure 4.

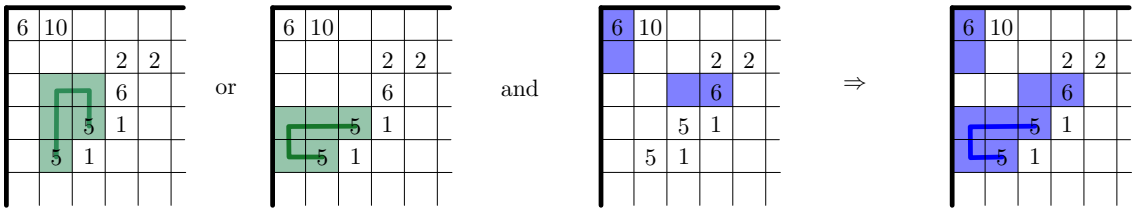

Figure 4. On the left, the 3 possible broken lines to link both '5'. On the right, only one is compatible with the occupied case next to the '6'.

Task 6. *Implement the combination and exclusion ideas to identify as many cases as possible which belong to the solution.*

Exclusion can give some information, only when the status of one of the case "near" to the clue has changed. To implement this idea, you may want to use a "dirty-flag" approach, and use exclusion only when needed.

Task 7. *Combine combination and exclusion with backtracking to improve your solver.*

Be explicit in your report of how you interlace backtrackings and inclusion/exclusion.

4. Extension: heuristics

To improve the efficiency of your solver, try to combine the systematic methods used above and some tricks and heuristics.

Task 8. *Improve your solver ! Discuss the efficiency of your ideas !*

Evaluation and trivia.

- You will get points for the correctness of the programs, points for the cleverness of the refinement of the program you designed, and points for the quality of the report and of the defense.
- The report must present concisely all the algorithms that have been implemented and explored; discuss their complexity and evaluate them on the provided data-sets. Defense will be 15 minutes long and will involve a live demo of the tool with data-sets possibly different from the ones provided and a discussion of the algorithmic choices.