STRAIGHT-LINE UPWARD GRID DRAWINGS MINIMIZING EDGE CROSSINGS

THIS PROGRAMMING PROJECT IS RELATED TO THE

LIVE CHALLENGE OF THE GRAPH DRAWING CONTEST (SEPTEMBER 2020)

LUCA CASTELLI ALEARDI (LIX, 2019-20)

ABSTRACT. The problem to solve will concern the computation of upward drawings of directed acyclic graphs on an integer grid. The main motivation of this project is to develop your own tool for efficiently computing such graph layouts, in order to participate to the Graph Drawing Contest 2020 (Live Challenge), on september 2020.

Requirements: basic knowledge of discrete maths and elementary geometry
Key words: graph drawing, computational geometry, graph theory.
Author: Luca Castelli Aleardi (`amturing@lix.polytechnique.fr`)

1. INTRODUCTION

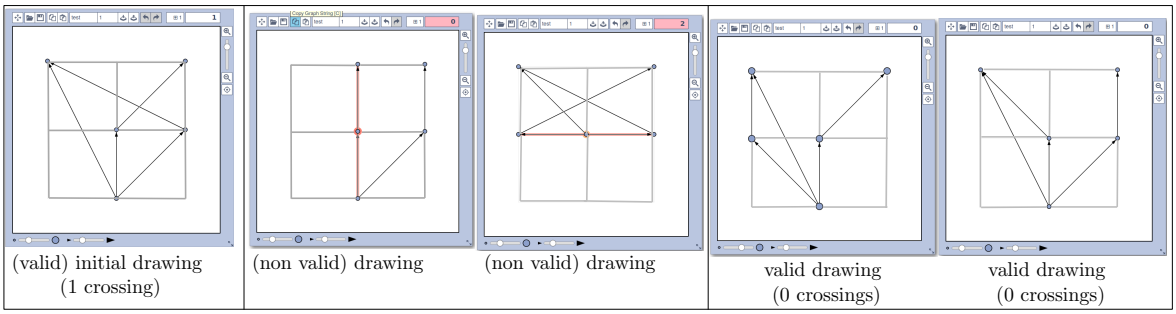

FIGURE 1. This pictures show a few grid drawings of a directed graph with 5 nodes and 5 edges, on a grid of size 2x2. (Left) A valid upward drawing with one crossing; (Center) two examples of drawings that are not valid; (Right) Two drawings that are valid and without crossings.

The next edition of the *Annual Graph Drawing Contest* shall take place in late september 2020 during the *28th International Symposium on Graph Drawing* (GD 2020, Vancouver, CA), one of leading conferences in the domain of graph drawing and network visualization. One of the two parts, the Live Challenge, consists in a sort of programming contest, where teams are asked to compute and submit nice layouts for some given classes of input graphs. In particular, in the automatic category teams are assumed to use their own algorithmic tool to compute layouts of graphs having up to a few thousands of nodes (the duration of the challenge is one hour).

In the 2020 edition (as for the 2019), the problem to solve will concern the computation of upward drawings of directed acyclic graphs on an integer grid (see below for more details). The main motivation of this project is to develop your own tool for efficiently computing such graph layouts, in order to participate to the Graph Drawing Contest 2020 (Live Challenge). Observe that there are no geographic restrictions for participating to this contest: teams (at most 3 people, one computer) can participate remotely in the automatic category, even not attending the conference in Vancouver.

1.1. Problem description.

The input of the problem consists of a graph that is *directed* (every edge is oriented in one direction) and *acyclic* (no directed cycles of edges), and a regular integer grid of size $width \times height$ (see Fig. 1 for an illustration).

A *straight-line upward grid drawing* is a 2D drawing of the input graph in which each directed edge is represented as a line segment such that the following conditions are satisfied:

(1) for each edge, the target vertex has a strictly higher y-coordinate than the source vertex;

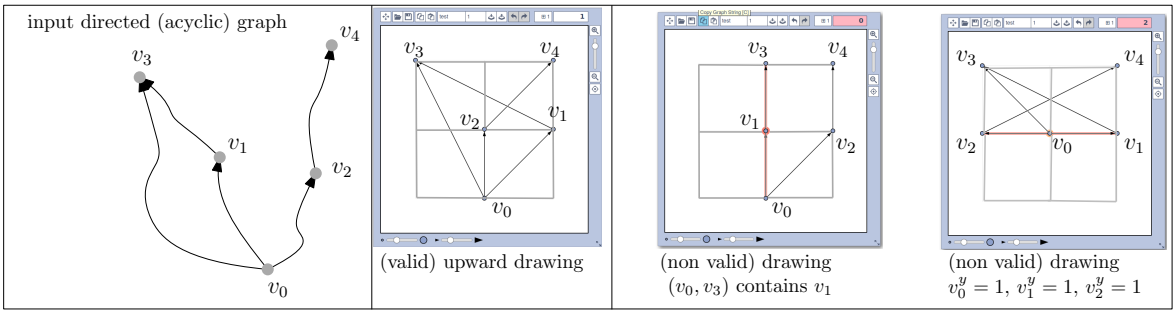

FIGURE 2. (Left) An input directed acyclic graph G. (Right) two non valid drawings of G on a grid of size 2×2.

(2) two edges (u, v) and (a, b) not sharing an endpoint may *cross*: but their intersection must be a single interior point of the image segments corresponding to (u, v) and (a, b);
(3) the graph is embedded on the input grid: the x-coordinate and y-coordinate of the vertices must be integers on a grid of size $[0..width] \times [0..height]$, where *width* and *height* are two integer parameters provided as input of the problem.

A grid drawing satisfying the constraints defined above will be referred to as *valid* drawing (see the pictures in Fig. 2 for an illustration).

The goal is to compute a valid straight-line upward grid drawing where the number of edge crossings is as small as possible.

1.2. **Input and output format.** Your program must take as parameter the input graph stored in `JSON` format: a JSON format contains a list of n vertices (also called *nodes*), numbered from 0 to $n-1$, each provided with x and y integer coordinates. Then it contains the list of oriented edges (also called *arcs*), represented with a pair of integer indices (the `target` and `source` vertices of the edge). This JSON file also contains two attributes `width` and `height` (the size of the grid).

The output of your program is a graph encoded in JSON format: the x and y coordinates encoded in the output JSON file are the ones computed by your program.

Observe that some (small) input graphs [1] are already provided with an initial drawing, while larger graphs (having thousands of vertices) are not provided with an initial embedding (all vertices are placed at $(0, 0)$, so the initial drawing is not valid).

1.3. **Submission requirement.** Your submission should include:

- your report (pdf file),
- the source code of your program (provided with a `README.txt` file explaining how to compile and run your program, if needed),
- the list of updard drawings (in `JSON` format), computed by your program corresponding to the input datasets provided with this project.

2. TASKS

As mentioned above your program must take as parameter the input graph stored in `JSON` format, and output a valid upward drawing (stored in JSON format). Your program should also be able to compute the number of edge crossings in the drawing.

For the sake of simplicity, we provide `Java` code allowing to read/write `JSON` files and to handle directed graphs.

You are free to choose your favourite programming language (`C/C++, Java, Python`, ...), but we strongly suggest to use the Java code provided with this project. Dealing with 2D layouts and input/output operations is a time consuming task: you are allowed to develop your own code, but no additional points will be assigned for this task.

In the case you make use of the provided Java code, we suggest to solve the tasks listed below, by completing the class `UpwardDrawing`. Feel you free to adopt different strategies and heuristics. At the end, the goal is to obtain a (time) efficient tool that computes a valid drawing with few crossings.

2.1. **Task 1: Checking the validity of a drawing.** The first step consists in checking whether a given drawing (a graph in `JSON` format, together with x and y coordinates of its vertices) is valid

[1] Input graphs are available for download on the web page of this project.

or not and to evaluate its number of crossings. The evaluation will take into account the efficieny of the algorithms and data structures adopted in your solution.

Question [4-6 points]. From the implementation point of view, you are asked to complete

- the method `isValid()` that checks whether a given graph (provided with geometric coordinates for its vertices) defines a valid drawing.
- the method `getCrossings()` that computes the number of edge crossings.

2.2. **Task 2: computation of a valid drawing.** You are also asked to compute a valid drawing (possibly having many crossings). Several solutions are possible: for instance, you can first compute a *topological ordering* (also called *topological sort*) of the graph with a DFS traversal, in order to guess an initial location of the vertices.

Question [6 points]. From the implementation point of view, you are asked to complete

- the method `computeValidInitialLayout()` that computes a valid drawing: the vertices are assigned x and y coordinates satisfying the prescribed requirements.

3. Minimizing the number of crossings

You are now asked to improve the drawing computed above (which is assumed to be valid) in order to reduce the number of edge crossings. We propose to make use of two simple heuristics (described below), but feel free to develop your own ideas and solutions.

3.1. **Task 3: Force-directed layouts (spring embedding model).** An important paradigm for computing graph layouts consists in implementing the so-called *spring embedding* model: vertices can be seen as particles of a physical system that evolves under the action of forces exerced on the vertices. For instance, in the spring-electrical model introduced by Fruchterman and Reingold in [1] there are attractive forces (between adjacent vertices) and repulsive forces (for any pair of vertices) acting on vertex u, which are defined by:

$$F_a(u) = \sum_{(u,v)\in E} \frac{\|\mathbf{x}(u)-\mathbf{x}(v)\|}{K}(\mathbf{x}(v)-\mathbf{x}(v)),\ F_r(u) = \sum_{v\in V, v\neq u} \frac{-CK^2(\mathbf{x}(v)-\mathbf{x}(u))}{\|\mathbf{x}(u)-\mathbf{x}(v)\|^2}$$

where the values C (the strength of the forces) and K (the optimal distance) are scale parameters (given as input constants). The layout computation proceeds in an iterative manner, computing at each step the displacement of each vertex u, that depends on the forces exerced on u. The algorithm terminates after a given number of iterations: the resulting layout corresponds to an approximation of a local minimum of the sping energy. More details on the implementation of spring embedders can be found in [2].

The spring embedder paradigm is rather flexible and can be adapted to deal with upward grid drawings: for instance, it could be possible to add new repulsive forces between neighboring edges, in order to reduce the number of crossings or to keep fixed the y-coordinates of some (well chosen) vertices.

Question [5 points]. From the implementation point of view, you may complete

- the method `forceDirectedHeuristic()` that run a spring embedder in order to get a valid drawing with a smaller number of crossings.

3.2. **Task 4: local search heuristic.** A further way to reduce the number of crossings consists in performing a *local search heuristic*. For instance, one can first compute the edges that most contribute to the number of crossings and try to move one of their endpoints (or both) to a new grid location that makes decreases the number of crossings. Vertices should be moved while maintaining a valid upward drawing.

Question [5 points]. From the implementation point of view, you may complete

- the method `localSearchHeuristic()` that performs a local search heuristic in order to get a valid drawing with a smaller number of crossings.

4. Evaluation criteria

Code evaluation. Your code will be evaluated according to the following criteria:

- the *clarity* of the provided code (your code should be well organized and commented, and include a clear definition of input and output parameters for all methods, according to Javadoc speficications);
- the code should be *correct*: it should compile and return a valid upward drawing for each input graph;
- your code should be *efficient*: we will evaluate both the runtime efficiency (elapsed time) and the number of crossings of the resulting (valid) drawings;

Overall project evaluation. The evaluation of this project will take care of several criteria, including:

- the quality of the oral presentation (the use of slides is strongly suggested);
- the quality of the report: the report should include a clear presentation of the results obtained with your programs (a short descrption of experimental results and runtime performances must be included);
- submission guidelines should be respected (see Moodle page for the PI);
- submission deadlines **must** be respected.

5. Appendix

5.1. **Representing directed graphs and visualizing graph drawings.** The java code provided with this project allows you to read the input graph (in JSON format, stored as a text file) using the library `TC` (developed at Ecole Polytechnique, see the website of the course INF311). You can run the class `ComputeUpwardGridDrawing` (with a JSON file as input parameter) to test your code and get a 2D layout of the resulting upward drawing.

References

[1] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw., Pract. Exper.*, 21(11):1129–1164, 1991.

[2] Yifan Hu. Efficient, high-quality force-directed graph drawing. *The Mathematica Journal*, 10(1), 2006.