Ecole Polytechnique

CS Project

INF 421

Straight-line upward grid drawings minimzing edge crossings

Author:
Farouk Khlifi
Adrien Lemercier

Project author:
Luca Castelli aleardi

February 3, 2020

Abstract

The problem we deal with in this paper concern the computation of upward drawings of directed acyclic graphs on an integer grid. We create methods to check the validity of a given drawing, to create a valid drawing from a given graph, and finally to diminish the number of edge crossings with two different heuristics.

1 Introduction

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* × *height*.

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;
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. if two edges (u, v) and (w, v) share a vertex v, then the intersection of their images must be a single point (the image of v);
4. 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.

In what follows we will use the following notations :

- n is the number of nodes of the graph,
- m is the number of edges,
- each edge has an *origin* and a *target*.

2 Checking the validity of a drawing

We have defined the method isValid() in the class UpwardDrawing that checks whether a given drawing (a graph provided with geometric coordinates for its vertices) defines a valid drawing.

Our code does basically the following things :

1. Checking that all points belong to the grid (i.e. their coordinates are not out of range) and that two of them are not at the same place.
2. Checking that all the edges go upward.
3. Checking that there is no forbidden crossings between edges. In order to do so, we use the fact that this statement is equivalent to : on each edge, there is no non-isolated node.

3 Computing the number of edge crossings

We have defined the method getCrossings() in the class UpwardDrawing that computes the number of edge crossings in the graph.

We had to define first the method intersect() that determines whether two edges intersect or not. But we have a more efficient algorithm than just checking all the $\binom{m}{2}$ pair of edges.

Indeed, we noticed that if an edge has an origin "higher" than the target of an other, then they can't intersect, so there is no need in taking into account this pair of edges. This observation leads to the following algorithm :

1. Arranging all the nodes in a list corresponding to the y-coordinate of their origin (there are height + 1 such lists).
2. We scan all the nodes from the bottom to the top, and from the left to the right. For each node, we consider all the edges that have this node has their origin. For each edge (a, b), we consider the edges that have their origin in $[y_a, y_b - 1]$ (for the horizontal line $y = y_a$: only the nodes that are on the right of a i.e. $x > x_a$).

By doing so, we reduce the number of calls to the method intersect(). The extend of this reduction depends on the graph itself.

For the graphs data...

!!! If three lines meet in one point, we count it as three intersections and not one.

4 Computation of a valid drawing

We have defined the method computeValidInitialLayout() in the class UpwardDrawing that computes a valid drawing of the graph.
It is a two-steps algorithm.

4.1 Step 1 : computing the height of each node

We define the height of a node *a* as the length of the longest path that finishes at node *a* in the graph. We call a node a root if no edge has this node as its target. The roots are exactly the nodes of height 0. There is at least one root in the graph we consider in this problem because they are acyclic graphs.

We compute the height of each node and store it as the tag (an attribute of the class Nodes) of the node with a generalized depth-first search algorithm.

More precisely :

1. We create a stack containing all the roots of the graph in a some order (we just scan all the nodes of the graph to do it). All the tags are initialized at 0.
2. For the node *a* at the top of stack, we scan its successors in some order.
   - if the tag of a successor is lower or equal to the tag of *a*, we set this value to 1 + (tag of a) and add it at the top of the stack.
   - if the tag of a successor is strictly greater than the tag of *a*, we don't do anything, and we consider an other successor of *a*.
3. once all the successors of a node *a* in the stack have been considered, we pop this node.
4. the algorithm stops when the stack is empty.

This algorithm finishes and at the end, each node has a tag corresponding to its height in the graph. Indeed,

1. at each moment, the tag of any node a corresponds to the length of an existing path going to a (loop invariant).
2. this implies that the tag of a node is always lower or equal than its actual height.
3. a node can not stay in the stack for always : it will be popped once (it can be added after again). This can be seen by induction on the length of the longest path starting at a given node.
4. one can prove by induction on the height of a node that there is a moment when its actual height will be set as its tag. We use crucially the fact 3. for that.
5. each time a node is added to the stack, its tag increases by at least one. So it will be added at most its height times.
6. the previous fact and 3. implies that the algorithm finishes. Thanks to 2. and 3., we can claim this algorithm is correct.
7. 4. also implies the complexity of this algorithm is $O(n \times h)$ where h is the maximal height in the graph.

4.2 Step 2 : computing a valid drawing

Once we have the tags of all the nodes, we can compute a valid drawing for this graph.

Here is the key fact : if (a, b) is an edge of the graph, then the height of b is strictly greater than the height of a. So if we put the nodes on the grid such that the higher their height the higher their y-coordinate, we are sure to have a upward drawing. The only concern is about the correctness of this drawing.

We compute $k = \lfloor \frac{height}{tag_{max}} \rfloor$. We place randomly the roots in the k lines at the bottom, then in the k following lines the nodes whose height is 1 and so on. Each time we place a node, we just make sure no other node is at the same place.

At the end, we check that there is no non-isolated node on an edge, and if there are, we move problematic nodes. More precisely : we go through all the nodes in some order. If there is any problem with a node (an other node is at the same place, or it is on an edge, or one its edges has a non isolated node on it), we move it randomly in the domain allowed by its height, until there is no problem with this node anymore. At the end, our drawing is correct (a node without any problems can't have a problem at the end, because each time we move a node we don't create any new problem, we only solve them).

Remark : there is an obvious weakness in this algorithm. If there are too many nodes at a given height, the corresponding domain may be full and the algorithm doesn't work. But let's notice one thing : the bigger the number of nodes, the harder the task, height and width being fixed. It can even be impossible, for example if a node has a height strictly greater than the parameter height. In practice, on the given dataset, our algorithm works, but one could find cases where a valid drawing exist but our algorithm can't compute it. Those specific cases seemed really hard to solve.

Experimental performance : on the graph graph6.JSON (118 nodes, height = width = 100), the algorithm has computed a valid drawing in less than one second. Only 5 nodes had a problem, and they have been moved only once. This shows that for values of height and width not too small, this algorithm works well, because the probability of being on an edge is very small.

5 Minimizing the number of crossings : Force-directed layouts (spring embedding model)

We generalize here ideas presented in the following articles :

[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.

We define an attractive force F_a as :

$$F_a(x) = \sum_{y \leftrightarrow x} K \left\| y - x \right\| (y - x)$$

and a repulsive one, F_r as :

$$F_r(x) = \sum_{y \neq x} -K \frac{y - x}{\|y - x\|^2}$$

where $K = \sqrt{\frac{height \times width}{n}}$.

The idea is the following : we compute for each node the total force that applies at that node, and then move the node in that direction. Of course, we allow only legal moves : the nodes have to have coordinates in $[0...width] \times [0...height]$, the edges have to go upward and so on. We execute those moves a defined number of times.

6 Minimizing the number of crossings : local search heuristic

We have defined the method localSearchHeuristic(int c) in the class UpwardDrawing that diminishes the number of edge crossings of a given correct drawing.

The idea is quite simple : we identify the most problematic node (i.e. the one whose the sum of the number of intersections on its edges is the highest). Then we find range of y-coordinates that are allowed for this node knowing the positions of its successors and predecessors. We pick randomly a position in the domain $[0...width] \times [y_{min}...y_{max}]$ and wonder whether the new drawing would be better or not, meaning that it is correct and has a strictly lower number of crossings. If it is the case, we move the node to this new position. If it's not, we try again (at most c times in a row for a given node. If it fails c times in a row, we will never look at this node again : it is "eliminated"). We do it again and again until all the nodes are "eliminated".

Why it is not that bad :

- At each move, the total number of crossings decreases strictly. The situation can only improve with time.
- It is natural to move the most problematic node first : we focus where the problem really is.

- It is hard to know where we should place this node. So we just let randomness explore the possibilities for us, and we pick it if we are satisfied with it.

- We can set a high value of parameter c, since our implementation is effective, giving better results (we are more likely to find a better place if any at each step).

Why it stops : Let's define $T \in \mathbb{N} \cup \{\infty\}$ be the total number of times we pick a new position randomly, and $\{Q_n\}_{0 \leq n \leq T}$ be the total number of crossings plus the number of nodes that are not eliminated yet, at time n (after we picked randomly n positions). Then :

- $Q_{n+c} < Q_n \quad \forall\ 0 \leq n \leq T - c$. This is because if we don't move the most problematic node at time n c times in a raw, then it will be eliminated (so at last at time $n + c$). So we have this inequality is the two cases.

- $Q_n \in \mathbb{N} \quad \forall\, n \leq T$.

Those two properties imply that $T < \infty$, meaning that the algorithm stops. And we are sure that the output is at least as good as the input. We can even deduce the upper-bound $T \leq c \times Q_0 \leq c \times (n + \binom{m}{2})$, giving an upper-bound for the worst-case complexity.

Effectiveness in practice : on a drawing of a graph with 39 nodes in a 50 by 50 grid (graph4.JSON from the dataset), the number of crossings went from

- 188 intersections to 85 in 0.08s (c=4)

- 170 intersections to 50 in 0.10s (c=10)

- 172 intersections to 58 in 0.31s (c=100)

- 175 intersections to 60 in 0.91s (c=1000)

- 125 intersections to 46 in 6.44s (c=10000)

Note that we can also use the method twice or more times to improve the improved result.