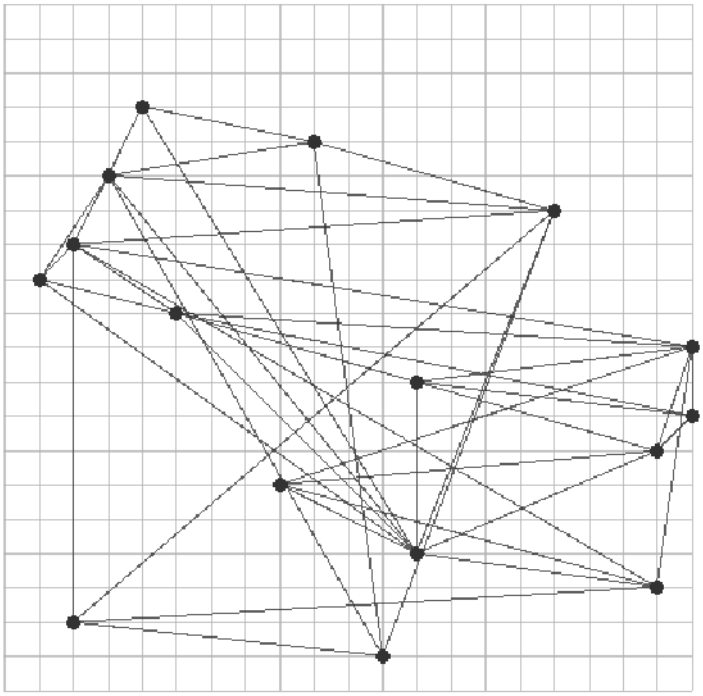


Graph Drawing Contest 2020

Ilias Saghir, Salaheddine Elkadiri

February 2020

Contents

1 Introduction

The next edition of the Annual Graph Drawing Contest will take place in late september 2020 during the 28th International Symposium on Graph Drawing (Vancouver, CA). One of the two parts, the Live Challenge, is a programming contest, where teams are asked to compute and submit layouts for some given classes of input graphs. 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, all within an hour.

2 Problem Description

Given a collection of input graphs that are directed and acyclic, a straight-line upward drawing is a drawing in which each directed edge is represented as a line segment such that the target vertex has a strictly higher y-coordinate than the source vertex. Two edges are said to cross if they do not share an endpoint, but the interiors of their images intersect. We are interested in computing grid drawings: the x-coordinate and y-coordinate of the vertices must be integers on a grid of size [0..width]x[0..height], where width and height are two parameters provided as input of the problem.
The goal in this project is to compute a straight-line upward grid drawing which minimizes the number of edge crossings.
In general, computing the crossing number of a graph G, which is the minimal number of crossings on a plane drawing of G, is a very famous problem in Graph Theory, that was shown to be NP-Complete by Garey and Johnson in 1983. In our case, we will be trying to compute an upward drawing with as little crossings as possible. For that, we will first compute an initial layout, and then proceed to reduce the number of crossings on it, using two different heuristics : a *spring embedding model* and a *local search heuristic*. We will also need to define two functions in order to check the validity of a drawing and compute the number of crossings. Our coding is done in Java.

Algorithm 1 Main Problem

Input : An acyclic, oriented graph $G = (V, E)$ and a grid of size $[0..width] \times [0..height]$

Output : Coordinates for each node $v \in V$ that form a valid upward drawing of G with minimal crossings.

3 Validity and number of crossings

3.1 Checking the validity of a drawing

A valid upward drawing is a drawing that meets the following conditions:

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

2) two edges that share an endpoint can only cross in that point

3) two edges that do not share an endpoint can cross but only in a point in the interior of their images

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.

As a first task in our project, we were asked to design an algorithm that checked whether a given drawing, that is, a list of vertices with their coordinates and a list of edges, was a valid upward drawing. A possible solution is to go through all edges and for each edge $e \in E$: check whether (1) is verified, and then for each $e' \in E \setminus \{e\}$ check whether e and e' cross, and if they do so correctly. Testing if an edge verifies (1) and whether two edges verify conditions (2) and (3) can be done in constant time, thus the validity check as defined above can be done in $|E|(|E|-1) = O(|E|^2)$ operations.

Algorithm 2 checkValidity()

Input : An acyclic, oriented graph $G = (V, E)$ provided with coordinates
Output : A boolean indicating wether the drawing is valid

```
S = ∅
for e ∈ E do
  if e does not verify (1) then
    return false
  end if
  if e overlaps with any edge v ∈ S then
    return false
  end if
  S = S ∪ {e}
end for
return true;
```

3.2 Computing the number of crossings

In order to compute the number of crossings of a given drawing, we go through every (unordered) couple (e, e') of edges, and test for a crossing. The algorithm is the following :

Algorithm 3 computeCrossings()

Input : An acyclic, oriented graph $G = (V, E)$ provided with coordinates
Output : An integer indicating the number of crossings

```
c = 0
for e ∈ E do
  mark e as visited
  for e' ∈ E do
    if e' is not visited and e and e' cross then
      c = c + 1
    end if
  end for
end for
return c;
```

We mark edges to avoid counting the same crossings twice. Testing if two edges cross can be done in constant time, and we have two nested for loops on the whole set of edges, thus the time complexity of our algorithm is $O(|E|^2)$. In our Java code, we tried to implement an efficient static method in the class Edge to test for crossings by avoiding to manipulate floats and doubles, and used a determinant based method that only operates on integers.

4 Computing a valid initial layout

4.1 Computing a topological sorting

In order to compute a valid drawing of an input graph G, it is useful to sort its vertices in a manner that facilitates verifying the condition (1) on y-coordinates. A topological sorting of a graph is a sorting in which no node appears before its predecessors, and this makes it easier to set its y coordinate strictly larger than that of all its predecessors, it also reduces the number of edges that we have to handle when placing a new node. In order to get a topological sorting of G's vertices, we use a depth-first search algorithm, for that we will need to define two methods :

Algorithm 4 TopoSort(Node u), a recursive method used by the topological sort

Input : A node u, an empty stack S

```
mark u as visited
for v ∈ u.successors do
  if v is not visited then
    TopoSort(v)
  end if
end for
S.push(u)
```

Since the graph is not necessarily convex, we still need to make sure to include all the nodes :

Algorithm 5 Sort()

Input : A graph $G = (V, E)$, an empty stack S
Output : A topological sorting of G stored in S

```
for u ∈ V do
  if u is not visited then
    TopoSort(u)
  end if
end for
return S;
```

The time complexity of the topological sort is that of a DFS $O(|V| + |E|)$

4.2 Valid initial drawing

We place node after node in the topological order, each time making sure the new node's spot is unoccupied and keeps the drawing valid. In order to ensure this, we make sure that the node's y-coordinate is strictly larger than that of all its predecessors, (which are already fixed since we followed the topological ordering), and that the edges that link the node to its predecessors do cross incorrectly with each other or with other existing edges, and that's all the edges we need to worry about since the successors have not been placed yet.

ComputeValidCoordinates is a function that attributes a valid position to the node u, taking into account *edges*, the set of already drawn edges. In theory, such position is not guaranteed to exist, and it depends a lot on the coordinates of the previously treated nodes, as well as the size of the grid. But if we suppose that such position always exists, which in practice isn't far fetched, then "the current upward drawing is valid" becomes a loop invariant. This is guaranteed by the fact that we proceed in a topological order : a node will always have

Algorithm 6 computeValidInitialLayout()

Input : A graph $G = (V, E)$, and a topologogical ordering S of its vertices
Output : A valid upward drawing (coordinates for each node)

```
edges = ∅
for u ∈ S do
  ComputeValidCoordinates(u,edges)
  for v ∈ u.predecessors do
    add (v,u) to edges
  end for
end for
return S;
```

a strictly larger y than its predecessors and ComputeValidCoordinates makes sure that conditions (2) and (3) are respected. The algorithm thus returns a valid upward drawing.

In our implementation, we compute a valid position for a node u this way : we first set the y coordinate of u to $max\{v.p.y | v \in u.predecessors\} + h$ where $h \in \mathbf{N}^*$ is the vertical step (the bigger h, the bigger the drawing), we then check all x-coordinate values in a random order until finding a valid position, otherwise we increment the y-coordinate by 1 and repeat the x-coordinate search.

5 Minimizing the number of crossings

5.1 Local search heuristic

In order to reduce the number of crossings on our previously computed drawing, we perform a local search heuristic that consists of repeatedly computing the edge that has the most crossings and moving one of its nodes to a better location. For that we use a priority queue Q where we store the edges in decreasing number of crossings.

Algorithm 7 localSearchHeuristic()

Input : A graph $G = (V, E)$ with an initial embedding
Output : A valid upward drawing with less crossings

let Q be a priority queue where all the edges are stored in decreasing number of crossings
int i = 0
while i <n_iterations **do**
  (u,v) = e = Q.poll()
  Remove all edges going to/from v from Q
  Move v to a "better" valid location
  Compute the crossings for the new edges going to/from v
  Add the new edges to Q
  i=i+1
end while

Once again, we chose the same random based method for computing the new location for the node v, this means the new location is not necessarily better than the old one, but since the initial position was also assigned "randomly", and since it caused a lot of crossings, then it was probably a rare case of bad luck, which is why we can hope that the new randomly assigned x-coordinate will decrease the number of crossings. Note that the difference with the previous algorithm is that if we go through all x coordinates without finding a valid position, we just leave it as it was, and move on to the following edge in the priority queue.

5.2 Spring embedding model

Another heuristic for reducing the number of crossings is the spring embedding model, that introduces attractive and repulsive forces between nodes as if they were physical particles. We introduce the two repulsive and attractive forces that were suggested :

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

In addition to these forces, we added a new repulsive force between neighboring edges in order to avoid overlapping edges and reduce the number of crossings, to make sure that condition (1) on y-coordinates is also respected, we also added a condition on the update of the node's coordinates : In a given iteration, we only move a node if its new position doesn't become too close (in terms of y) to that of any of its neighbors. Note that during the whole algorithm, the coordinates are floats, and we reconvert them all to the closest integer at the end of the while loop, which means there's a risk that two nodes might have

the same integer y coordinate if they were too close. step is a parameter that is updated in every iteration in order to make the algorithm converge.

Algorithm 8 forceDirectedHeuristic()

Input : A graph $G = (V, E)$ with an initial embedding
Output : A valid upward drawing with less crossings

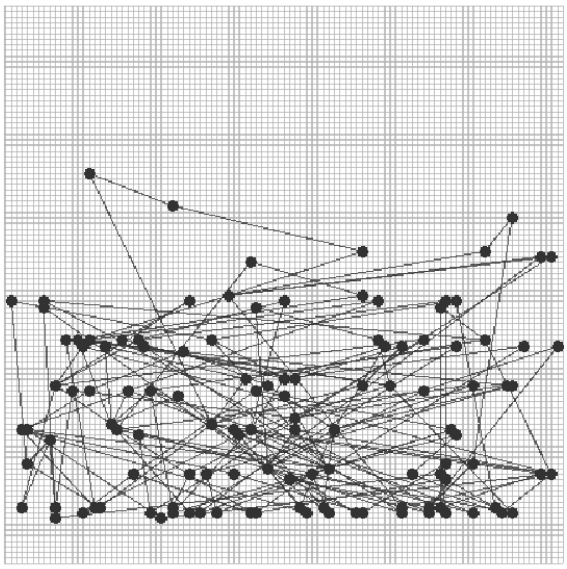
```
int i = 0
while i < n_iterations do
  for Node u ∈ V do
    u.force += F_a(u) (attractive force with neighbors)
    u.force += F_r(u) (repulsive force with all other nodes)
  end for
  for every couple of edges (e, e') ∈ E^2 do
    if e then
      and e' are neighbouring edges apply repulsive forces to the respective
      nodes
    end if
  end for
  for Node u ∈ V do
    if u.p.y + u.force.y is sufficiently far from v.p.y for every v ∈ N(u) then
      u.coords = u.coords + step * force/||force||
    end if
  end for
  update(step)
  i++
end while
Reconvert all coordinates to grid coordinates
```

6 Results and conclusion

Here's the result when we apply all the previous steps to Graph_06 :

```
Tools for the "Graph Drawing Contest 2020: Live Challenge"
Reading (undirected) graph in JSON format: D:\Users\Ilias\Documents\INF421\Graph Dra
        Reading nodes...done (119 vertices)
        Reading edges...done (166 edges)
Input graph loaded in main memory (119 vertices, 166 edges)
Reading width and height of the drawing area...done (100 x 100)
The input graph has an initial embedding
Compute a valid drawing with few crossings: Computing valid initial layout :
Initial layout computed in 0.0046894 seconds
Initial graph is valid
Initial number of crossings : 2251
Launching local search heuristic :
Local search time : 0.0308091 seconds
Graph after local search is : valid
Crossings after local search :1648
Now launching force directed heuristics :
Force directed heuristic time 1.1317749 seconds
Graph after force directed heuristics is : valid
Final number of crossings : 1487
Saving upward drawing to Json file: output.json ...done (119 vertices, 166 edges)
Setting Canvas size: 600 x 600
```

Here's the drawing :



The random choice for the x coordinate makes it easy to compute an initial drawing rapidly and efficiently, but it tends to create a lot of crossings. The two heuristics do a great job at eliminating many of them, but if we compare to the possible crossings number of 448 for graph 6, there's still a lot of room for improvement, both on the initial drawing method, and on the heuristics used.