ÉCOLE POLYTECHNIQUE

Report Of Project : Graph Drawing Contest 2020

Benxin ZHONG and Yiming QIN

2 février 2020

Abstract

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. 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.

In our project, we have defined the class `DirectedGraph` to store the graph. We defined a method to verify whether one layout of the graph is valid or not with the time complexity $O(mn)$. Then, we defined the method to generate an initial valid layout using topology sorting and random method, with the complexity $O(md+n)$.

In section 3, we developed the class `FDGraphDrawing`, in which we implemented the method to reduce the number of intersections based on the Force-Directed Drawing method. This method has a complexity $O(m+n)$ in each step. It terminates when the derived step length is blow a given tolerance.

In section 4, we developed two classes, `LocalSearchHeuristic` and `LocalHeuristicByNode`, which implement respectively the local optimization method of simulation annealing based on Markov progress, and the optimization of local traversal.

Key words : Directed Graph, Graph Drawing Contest, Force Directed Graph Drawing, Markov Process, Simulation Annealing.

CONTENTS

1 GRAPH STRUCTURE, VALIDITY, AND THE CROSSING NUMBER

Here, we study the layout of a directed graph : straight-line upward grid drawing. Our task is to generate a valid layout of this kind for a given directed graph, and then try to reduce the number of intersections in this layout. In this section, we are going to introduce the definition of this kind of drawing and then, give our methods of verifying the validity of a graph layout, of computing the number of intersections, and of generating an initial valid layout.

1.1 DEFINITION OF STRAIGHT-LINE UPWARD GRID DRAWING

Given a directed acyclic graph $G = (V, E)$, we note the number of its vertices $|V|$ by n, and the number of arcs (i.e., directed edges) $|E|$ by m.

A valid *straight-line upward grid drawing*(graphdrawing.org,) is a graph layout satisfying :

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 $[0..width] \times [1..height]$, where width and height are two integer parameters provided as input of the problem.

1.2 METHOD TO VERIFY THE VALIDITY

In the interface class `UpgridDrawing`, we have implemented the method `boolean checkValidity()` that checks the validity of our layout.

Firstly, we verify whether all the vertices are in the specified range, i.e. $[0,\dots,\text{width}] \times [0,\dots,\text{height}]$, to check the condition 4. This process has a complexity of $O(n)$.

Then, we invoke the interface `boolean isValid()` defined in the class `DirectedGraph`. This method iterates through the list of all vertices, and checks whether condition 1 is satisfied. Each iteration has a cost s_i, where $s_i = |\text{Succ}(v_i)|$, is the number of successors of vertex i. In this iteration, it also traverse the list of edges, to check whether the condition 2 and 3 are satisfied. Each iteration has a cost m.

Therefore, over method of checking the validity has a complexity $O(mn)$.

1.3 METHOD TO GET THE NUMBER OF CROSSINGS

We have implemented the method `int computeCrossing()` that returns the number of crossings in the layout. This method invokes the interface `int numGraphIntersected()` defined in the class `DirectedGraph`.

This functions has a complexity of $O(m^2)$ to verify all the pairs of edges $\{e_1, e_2\}$ whether e_1 and e_2 are intersected. Particularly, it does not just return the total number of crossings, but also stores in each edge e the number of intersections caused by this edge. This attribute can be used farther in the local optimization process.

1.4 TWO MAIN CLASSES AND CORRESPONDING INTERFACES

We defined the class `DirectedGraph` to deal with the structure of our graph. In this class, there are four members. The `vertices` of type `ArrayList<Node>` stores all the vertices in this graph, the `labelMap` of type `HashMap<String, Node>` maps a label to the vertex corresponding, the `edges` of type `ArrayList<Edge>` stores all the edges in this graph, and `level_list` of type `ArrayList<List<Node>>` stores the vertices based on their level that we classified.

Since we developed in the base of structure given in the file `src.zip`, there are several interfaces already implemented in the class `DirectedGraph` that are used to construct the graph from a `json` file, but are not related to our implementation, we will not introduce them.

Another useful class is `UpwardDrawing` in `Visual` package. This class stores a given directed graph with its layout, as well as the width and height information. This class is used as the interface that the users can invoke our methods. Each method that we have implemented has a corresponding interface in this class. The principal interfaces are :

- `boolean checkValidity()`, to check whether this layout is valid.
- `void computeValidInitialLayout()`, to generate a valid drawing if it is invalid.
- `int computeCrossings()`, to return the number of crossings.
- `void forceDirectedHeuristic()`, to process the force-directed drawing to reduce the number of crossings.
- `void localSearchHeuristic()`, to use the simulation anealing method to reduce the number of crossings.
- `void localSearchHeuristicNode(int tol)`, to use the local optimization process on vertices to reduce the number of crossings.

2 GENERATION OF A VALID GRAPH

In this section, we talk about the method of generating an initial valid layout.

The process of generating is mainly in two steps. Firstly, we classify the vertices topologically, and label each of them with a attribute level. Secondly, we arrange the vertices having the same level.

2.1 Classification of vertices

Firstly, we define inductively the attribute level in the vertices set, and therefore the classification of vertices based on their level :

DEFINITION :
Let L_k denotes the set of all vertices whose level is k.
For a vertex $v \in V$,

1. If $\text{Pred}(v) = \emptyset$, we define that $\text{level}(v) = 0$. i.e., $L_0 = \{v \in V | \text{Pred}(v) = \emptyset\}$.
2. Have defined L_k, we let $L_{k+1} = \{v \in \text{Succ}(L_k) | \text{Pred}(v) \subset \bigcup_{i=0}^{k} L_i\}$.

We can easily prove that this definition is rational. i.e., $\forall v \in V$, $\exists! k \in \mathbb{N}$, s.t. $v \in L_k$.

Since the definition is inductive, we can do the process of classification also inductively. In the method `void nodeClassifier()` defined in the class `DirectedGraph`, we firstly find all vertices of level 0 by traversing the list of vertices and labeling v as level 0 if `v.successors` is empty, and add v to the list `level_k_list`. This process has a cost $O(n)$.

Having labeled all the vertices of level from 0 to k and stored all the vertices in the list `level_k_list`, we traverse all the successors of vertices in this `level_k_list`, and find the vertices of level $k+1$ by definition. This iteration has a cost of time $\sum_{v \in L_k} \sum_{w \in \text{Succ}(v)} |\text{Pred}(w)|$.

Therefore, this process has a total complexity of $O(n) + \sum_{v \in V} \sum_{w \in \text{Succ}(v)} |\text{Pred}(w)| = O(md + n)$, where d is the degree of G, i.e., $v = \max_{v \in V} \{|\text{Succ}(v)| + |\text{Pred}(v)|\}$.

2.2 Random allocation in each layer

As it is enough complicated to ensure the others conditions of validity, we decided to applique the method of random allocation. That is to say, for the ith point in layer k which possesses l_k points in total, we will put it randomly in $x \in [(i-1) \times \text{width}/l_k, i \times \text{width}/l_k]$ with a uniform distribution. At last, we check the validity allocation given by this algorithms, where a `false` leads to a new generation and a `true` will return a valid allocation.

2.3 Corresponding interfaces

To generate the initial valid layout, you can invoke the interface `void computeValidInitialLayout()` defined in the class `UpwardDrawing`. This interface classifies the vertices, and generate a random allocation by invoking the function `allocateRandom()` in class `DirectedGraph` if the current layout is not valid.

The `allocateRandom()` function allocates each vertex a y value based on its level, and

then in each layer it calls the `private void distributeInLevelRandom(List<Node> level_n_list, int width)` function to allocate the x values of these vertices randomly.

3 Forced directed algorithm

In this section, we study the Force-directed drawing method (Hu,). In this method, the vertices of graph are considered as particles in a physical system that moves under the forces on them. We define that there are the attractive forces between the adjacent vertices, and there are repulsive forces between each pair of vertices. Then, we simulate the process that the vertices move in this force field, such that the *total energy*, i.e., the summation of squared forces, converges to the minimum, or a local minimum.

The pseudo-algorithm is provided in the paper (Hu,). In our project, to preserve the validity condition 1, we modified the algorithm, such that the vertices cannot move but only in its own layer. That means, the y coordinates are fixed, and they move only horizontally. We give our algorithm in Algorithm 1 on page 5.

But the process can break the other three conditions. In our project, we solve this problem in this way :

1. Ignore the condition 4. i.e., the x coordinate can be negative or can be above `width`.
2. After the iteration has finished, we search all vertices breaking condition 2 or 3. i.e., all vertices in on edge of which it is not an endpoint.
3. We move the each invalid vertex by 1, with the direction the same as that of the summation force on it.
4. Repeat the step 2 and 3, until that there are not vertex breaking condition 2 or 3.
5. Shrink the whole layout, such that all x coordinates are in the given range.

3.1 Modification of parameters

In this algorithm, there are several parameters which need to be given.

— The value of C and K is not important. So we set $C = 1$ and $K = 1$. (Hu,)
— Step is given by a an updating function to avoid tripped by local minimums. (Hu,)
— Tol is set initially to be 1 which showed a good performance. Another converge condition is that the algorithm can loop no more than 4000 times to prevent endless loops.

3.2 Complexity and results

In this process, we call an iteration in the outer loop a *step*. Each step contains a loop of n iterations, and each interation has a complexity

$$T_v = O(d_v + n)$$

Algorithm 1 Generate a graph by FD method

```
Input: max_step_num, initial_step_length, tol
  energy_old ← ∞
  step_length ← initial_step_length
  for i = 0, 1, 2, ..., max_step_num do
    if step_length < tol then
      break loop
    end if
    energy_new ← 0
    for v ∈ vertices do
      force(v) ← forceTotal(v)
      energy_new ← energy_new + (force(v))
      move(v, force(v), step_length)
    end for
    step_length ← updateStepLength(energy_old, energy_new, step_length)
    energy_old ← energy_new
  end for
  Make the layout valid
```

where d_v is the degree of the vertex v, i.e. $d_v = |\text{Succ}(v)| + |\text{Pred}(v)|$.

Each step contains also a process of updating, which has a constant complexity. Thus the total complexity of each step is

$$T_i = \sum_{v \in V} T_v = \sum_{v \in V} O(d_v + n) = O(m + n^2)$$

Then we need to estimate the number of steps we make. Based on our `updateStepLength` function, and the condition of termination, we have the number of steps is

$$N_{\text{step}} = \min\{\texttt{max\_step\_num}, \frac{\log(\texttt{tol/inital\_step})}{\log \alpha}\}$$

where α is a constant. Doing the upper bound estimation, we can take $N_{\text{step}} = \texttt{max\_step\_num}$, denoted by $M_{\max}$.

The process of makeing the layout valid has a complexity $O(n)$, where we do not give the proof. Therefore, the total complexity of the Force-Directed drawing algorithm is

$$T = \sum_{i=0}^{M_{\max}} T_i + O(n) = O(M_{\max}(m + n^2))$$

3.3 Class and interfaces

For the readability of the project, we created a new package called ForceDirected, which concludes a ForceDirected class. We added the necessary parameters such as energy, C, and K to the class, and implemented the function for reconstructing the graph.

The principal interfaces are :

— `GridVector_2 forceTotal(Node node)`, to calculate the force of a node in the forme of an vector.

— `void step()`, to calculate the forces, the energy, and to execute the move of every node for one time.

— `forceDirectedProcess(int max_step_num, double initial_step_length, double tol)`, to generate a valid graph with given parameters.

4 LOCAL OPTIMIZATION METHODS

We observe that for some graph, like graph 3 and graph 9, even the complexity of graph is not very high, FD algorithm can not give a satisfying result. That could be caused by the inner structure of the graph, who make edges intertwined driven by external forces. Therefore, we posed other more direct and intuitive methods to solve the problem, which is named as Local Search algorithm. We will call it LS algorithm in the following text.

4.1 INTUITION OF LS ALGORITHM

How to reduce the total number of intersections in the graph? The most direct idea is similar to the idea of the greedy algorithm. At each step, we find the edge (or point) with the largest number of intersections in the graph, and continuously reduce the number of intersections of this edge by moving the endpoints at both ends of the edge (or the point itself). We call this process local optimization, which is where the algorithm's name comes from. After that, we continue to cycle this process until the number of cycles exceeds a predetermined number of times, or until the convergence condition are met. At this time, the result is considered to be convergent and the program ends.

The core part of this algorithm is the local optimization process. We have also proposed two different algorithms. As can be seen in the results analysis later, their performance on different graphs is also different. Now we will introduce them.

LS ALGORITHM BY NODES First, let us consider the method of local optimization points. We define the number of intersections of a point as the sum of the intersections of an edge starting from that point or pointing to this point. In addition, we notice that edges are always pointing to higher position, so we need to restrict this point to only move on the x axis. Within the limits of these two, first, we find the point with the largest number of intersections, and we make it iterate through all the positions on the x-axis to accurately find the value of the local lowest point. Next we enter a new loop until the results converge, that is to say, until the time when the intersection change is less than a constant after one local optimization process of a certain edge.

We give our algorithm in Algorithm 2 on page 8.

LS ALGORITHM BY EDGES However, is the point a unit that is too small? Can this local optimization play a big role in the whole graph? Thus our second natural idea is to expand the local optimization to edges. The basic idea of this method is the same as the previous one, but it is not difficult to find a delicate point. If we want to find the lowest point of an edge by iterating through x axe, the time complexity of each loop is $O(\texttt{width}^2)$, which for the above one method is only $O(\texttt{width})$. When `width` is very large, it will be catastrophic. Therefore, we decided to implement a virtual annealing process using a Markov chain to approach the lowest point.

How to simulate such a Markov process? Our probability space is all positions on the x-axis of the row. Since the points and points cannot overlap, the transition matrix Q corresponding to the Markov process is defined as 0 at the occupied position of the row and the position of x itself. And other positions have the same probability. We also define an acceptance function R. R measures the possibility of moving from the current position to a position y. If the value of R is greater than $\frac{1}{2}$, it means that the number of intersections at this y is less than the current position, and the greater the probability of moving to next position is. If $h = \frac{u}{1+u}$, then R is defined as follows :

$$R(x,y) = \begin{cases} h(\exp(\frac{1}{T}(V(x) - V(y)))\frac{Q(y,x)}{Q(x,y)}), \text{ if } Q(x,y) \neq 0 \\ 0, \text{ if not.} \end{cases}$$

Then we process in the following way :

STEP 1 Initialize $X0$, which is the original graph (if the graph is not valid, call the generating valid graph function to make it valid)

STEP N Randomly select y according to $Q(X_n, y)$, and then randomly select U_{n+1} in $[0,1]$, if $U_{n+1} < R(X_n, y)$, then $X_{n+1} = y$, otherwise $X_{n+1} = X_n$

We give our algorithm in Algorithm 3 on page 8.

4.2 COMPLEXITY AND RESULTS

LS ALGORITHM BY NODES Here is the complexity of a local optimisation of a node which is given by the interface`process()` because the convergence number if not predictable.

The complexity of the whole process is executed for every element in x axe in which we calculate the total number of neighbor points' intersections. As the process of getting number of intersections of an edge consists of a loop of each edge in the graph, so its time complexity is $O(m)$. Thus we have the time complexity :

$$T = O(\texttt{width} \times mn)$$

Algorithm 2 Simulated Annealing

Input: `max_num_itr, tol, max_num_markov`
  for `i` from `0` to `max_num_itr` **do**
    `edge_max = (u,v)` ← the edge that causes the largest number of crossings.
    for `j` from `0` to `max_num_markov` **do**
      `w` ← `random_choose(u,v)`
      `T_w` ← `C*V_w(w.x)/ln(j)`
      `y` ← `random_choose([0,..,width] \ occupied_points)`
      `r` ← `random_double(0,1)`, obeying uniform distribution.
      if `r < R(x,y;T_w)` **then**
        `w.x` ← `y`
      end if
      if `T_u < tol && T_v < tol` **then**
        `break loop`
      end if
    end for
  end for

Algorithm 3 Node Traversal

Input: `tol`
  `num_crossing_change` ← inf
  while `num_crossing_change > tol` **do**
    `vertex_target` ← the vertex whose edges cause the largest number of crossings.
    Traverse all the non-occupied x coordinate values, and find `x_0` such that `vertex_target.x=x_0` makes the number of crossing minimal.
    `vertex_target.x` ← `x_0`
    `num_crossing_change` $gets$ the change of number of crossing after this loop.
  end while

LS ALGORITHM BY NODES As before, we consider first the time complexity of a local process which includes mainly `max_iter` loops for the corresponding edge. In this loop, all concerning operations use constant time. In this way, the local time complexity is :

$$T = O(\texttt{max\_iter})$$

The complete process has not only execute the local process, but also include an upgrading of the total intersection number (for every edge, its intersection number is calculated here too) and an upgrading of the edge who has most intersections. The time of the first upgrading given by the function `numGraphIntersected` is $O(m^2)$ because of two loops of edges. Then the second needs $O(m)$ on using the simplest find-max methode.

So in the end, the complexity of the whole processus is $\texttt{num\_edge\_process} \times (O(\texttt{max\_iter}) + O(m^2))$. We have :

$$T = O(\texttt{num\_edge\_process} \times ((\texttt{max\_iter}) + m^2))$$

One remark here is that as every time we need to recalculate the number of intersections of each edge, which may lead to a large time complexity.

4.3 CLASS AND INTERFACES

These two algorithms are in the LocalSearchHeuristic and LocalHeuristicByNode files under the LocalSearch package.

The principal interfaces of them both are :

— `void processEdge/processNode(Edge edge, double tol, int max_iter)`, to get on a local optimisation process.

— `void process(double tol, int num_edge_process, int max_iter)`, to generate a valid graph with given parameters.

5 COMPARISON OF THREE METHODS

Next, we will analyze the time complexity and the final performance of the three methods given in the above paper.

5.1 PERFORMANCE IN DIFFERENT GRAPHS

Graph number	Initial intersections	FD result	LS by nodes result	LS by edges result
1	1	1	1	1
2	1	0.22	0.91	0.19
3	157	0.34	0.19	0.06
4	390	0.17	0.33	0.36
5	180	0.06	0.08	0.09
6	447	1	1	1
7	3368	0.84	0.59	0.63
8	7108	0.60	0.88	0.80
9	1173	1	0.83	0.21
10	20302	0.97	0.99	0.57
11	102695	1	0.99	1
12	954832	1	1	0.73

Several remarks :

— In the column of every method, we give the proportion between the final intersection numbers and the initial intersection numbers.
— We display the generating results in the form of a table. Note that if the image is invalid at the beginning, we will call the previous generating method, which will cause the initial number of intersections of each image to be different, but after testing, such small differences in the image have little effect on the final algorithm result. So, for each graph, we will randomly use the number of intersections after the legalization process.
— In the algorithm, if the final number of intersections is greater than the initial number of intersections, the initial image is returned. So the result 1 indicates that the number of intersections is unchanged or rising.

Observing the table, we find :

1. The optimal result of more than half of the images is obtained by the LS by edges algorithm, which can show that this algorithm is generally effective ;
2. However, for Graph 5, the FD method is very useful which depends on its structure. We can see more intuitively these four methods in the figure 5 on page 11. With a good internal structure, the FD algorithm can effectively spread out intricate points, thereby obtaining a highly symmetrical result with few intersection points. The LS

by nodes method can greatly reduce the number of intersections through local optimization, but the LS by edges method is more effective than it. Nevertheless, the sole purpose of the LS by edges method is to reduce the number of intersections, so it does not consider the uniformity of the distance between points at all. Even its output looks more messy, but it does have fewer intersections.

3. For some images with very high complexity (such as Graph 11), or the internal initial structure is not optimistic (such as Graph 6), or small images which limit our randomization algorithms and the force directed algorithm (such as graph 1), none of the three algorithms can give satisfying performance.

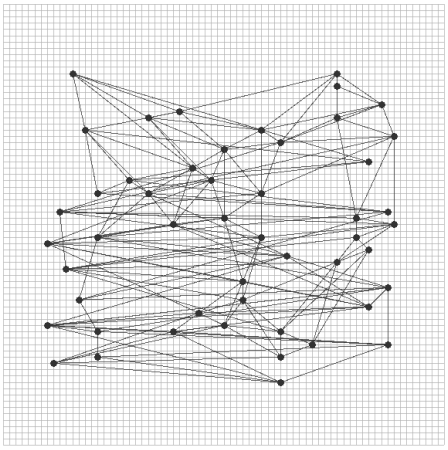

FIGURE 5.1 – Graph 5 initial output

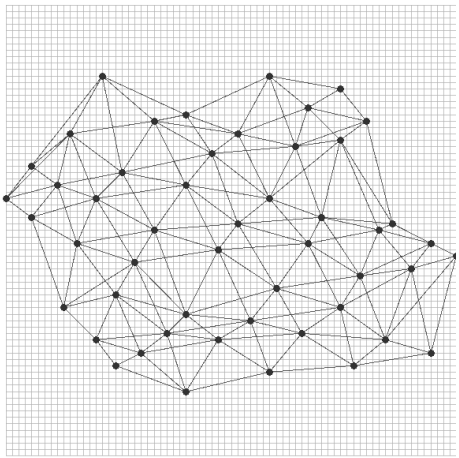

FIGURE 5.2 – Graph 5 after FD process

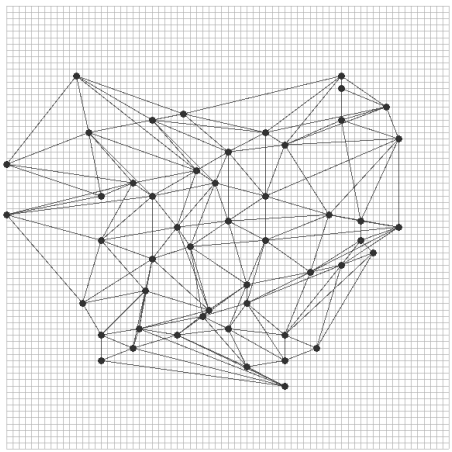

FIGURE 5.3 – Graph 5 after LS by nodes process

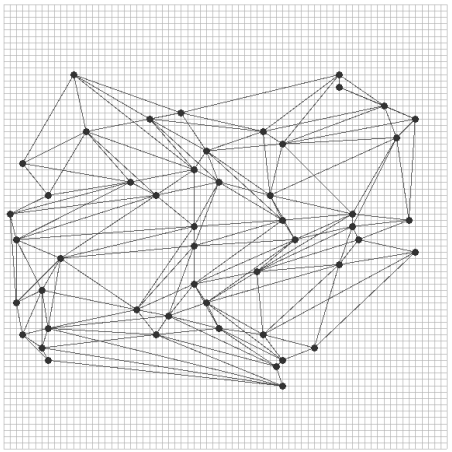

FIGURE 5.4 – Graph 5 after LS by edges process

5.2 TIME COMPLEXITY

We use a similar table to show the time complexity of these three algorithms in different graph. As our algorithms does not do operations in y axe so we will only look at the graph

width. We will also only choose 5 examples with very different dimensions.

Graph number	Width	Number of nodes	Number of edges	FD result (ms)	LS by nodes result (ms)	LS by edges result (ms)
2	20	16	42	8	5	366
7	100000	100	150	120	16386	7941
10	1000000	500	684	1197	190553	245671
11	1000000	1800	6961	10724	4232	2533482

The table tell us :

1. The time complexity increase of the FD algorithm is the most stable. Even for large graphs, it can be solved in a reasonable time, but its results may not be very good.
2. However, the LS by nodes algorithm increase quickly. But if, after a local optimization, the method detects that the results do not improve significantly, the program will end automatically like in Graph 11. It should be noted that although we also set the value of `tol` for LS by edge algorithm, but here the `tol` is used for step length which will by reduced if the number of intersections augment. Due to the randomness of this algorithm, the chance of not having improvement continuously in several local optimization is very low, so Graph 11 will converge quickly only for LS by nodes algorithm.
3. The LS by edges algorithm always has a very high time complexity, and this complexity also gives it more outstanding performance.

REFERENCES

graphdrawing.org. (2019). *Graph drawing contest.* Consulté le 2020-12-30, sur `http://mozart.diei.unipg.it/gdcontest/contest2020/contest.html`

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