Force Directed Graph Drawing Algorithms

Elias Röger, Ekaterina Tikhoncheva

23.02.2015

1 Introduction

The drawing of graphs, that satisfy some predefined conditions, is one of the most important topics in different fields of application such as information visualisation or chips production. There are a lot of different techniques of drawing the „nice“ graphs. In this work we present two algorithms from class of force-directed graph drawing algorithms [**?**]. In the first section we consider the approach by T. Kamada and S. Kawai [**?**] from 1988 and in the second section the multi-scale algorithm by D. Harel and Y. Koren [**?**], which can be considered as extension of each force-directed graph drawing algorithm to deal with bigger graphs. In the last section we report the results of our implementations of both algorithms and compare them with the results from the papers.

2 Algorithm by Kamada and Kawai

We implemented an algorithm for drawing general undirected graphs from [**?**]. In the following we refer to this algorithm as KKA (Kamda Kawai algorithm).
KKA visualizes connected, weighted, undirected graphs as a 2 dimensional picture where the edges are drawn as straight lines between the vertices. The authors state objectives for their graph drawing algorithm:

1. uniform distribution of vertices and edges
2. preserve symmetric structures
3. low number of edge crossings

The first two objectives are more important for human understanding than the third one. Hence, the goal of KKA is to find a state where the vertices and edges are distributed uniformly. 2. and 3. often follow from that.
We can now describe an optimal drawing of a graph mathematically. We try to minimize the sum of the differences between the desirable (Euclidian) distances and the actual distances between all pairs of vertices. Now let $G = (V, E)$ be a graph with $n = |V|$ vertices. We define $p_1, \dots, p_n$ as the projections into the 2 dimensional plane of the vertices $v_1, \dots, v_n \in V$. The appropriate mapping $L : V \to \mathbb{R}^2$ is called a layout of a graph G. We now define the function

$$E(p_1, \dots, p_n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} \{|p_i - p_j| - l_{ij}\}^2 \tag{1}$$

which models the imbalance of our drawing. We define $(d_{ij})_{i,j=1,\dots,n}$ as the matrix of distances between all vertices and $(l_{ij})_{i,j=1,\dots,n} = L \times d$ is a scaled distance. $(k_{ij})_{i,j=1,\dots,n}$ is a weight

matrix and its entries are defined by $k_{ij} = K/d_{ij}^2$. Since our graphs are undirected d, l and k are symmetric.
We now try to find a local minimum of E. Consequently we are looking for p_m with

$$\frac{\partial E}{\partial p_m} = 0 \quad for\ m = 1, \ldots, n.$$

This yields $2n$ nonlinear equations. We set $p_m = (x_m, y_m)$ and solve the equation for every m by using the two-dimensional Newton-Raphson Method. The means we iterate

$$\begin{pmatrix} x_m^{t+1} \\ y_m^{t+1} \end{pmatrix} = \begin{pmatrix} x_m^t + \delta x \\ y_m^t + \delta y \end{pmatrix}.$$

and stop when $\Delta_m := \sqrt{\frac{\partial E}{\partial x_m}^2 + \frac{\partial E}{\partial y_m}^2}$ is smaller than a predefined ϵ for all m. The increments δx and δy are defined by

$$\begin{aligned} \frac{\partial^2 E}{\partial x_m^2}(x_m^t, y_m^t)\delta x + \frac{\partial^2 E}{\partial x_m \partial y_m}(x_m^t, y_m^t)\delta y &= -\frac{\partial E}{\partial x_m}(x_m^t, y_m^t), \\ \frac{\partial^2 E}{\partial y_m \partial x_m}(x_m^t, y_m^t)\delta x + \frac{\partial^2 E}{\partial y_m^2}(x_m^t, y_m^t)\delta y &= -\frac{\partial E}{\partial y_m}(x_m^t, y_m^t). \end{aligned} \tag{2}$$

This leads to the algorithm:

Algorithm 1 Kamada Kawai [**?**]

```
Compute the matrices d, l and k
initialize p_1, ..., p_n
while (max Δ_i > ε) do
    m = argmax Δ_i
    while Δ_m > ε do
        compute δx and δy
        iterate x_m = x_m + δx, y_m = y_m + δy
        compute Δ_m
    end while
    compute Δ_i for i = 1, ..., n
end while
```

3 A fast multi-scale algorithm by D. Harel and Y. Koren

The second algorithm we tested is the algorithm by D. Harel and Y. Koren [**?**] (further HKA), which also handle the problem of drawing an undirected graph $G = (V, E)$ with straight line edges.

The issue, the authors were dealing with, is the low speed of the most at the time force-directed drawing algorithms due to optimization of the quadratic energy function, which makes them almost inapplicable for the large graphs with more than 1000 vertices. To cope with the speed problem authors continued the approach from [**?**] and suggested new faster multi-scale algorithm.

The idea of the their algorithm is very simple: instead of consider the whole graph G at ones one builds an approximation sequence of coarse graphs $G^{k_1}, G^{k_2}, \ldots, G^{k_l},\ k_1 < k_2 < \cdots < k_l = |V|$ (*multi-scale representation of* G), and applies one of the simple force-directed drawing algorithm to draw those graphs nicely. Beatification of each scale provides a locally nice layout of G and sequence of all locally nice layouts approximates the nice layout of G. We refer to the paper [**?**] for theoretical background of multi-scale representation of a graph, locally and globally nice layouts and describe here only the algorithm presented by authors.

Algorithm

To build a coarse graph of a given graph G authors suggest to cluster vertices of G in k groups, so that the distance between vertices in the same group is minimized. This corresponds to the simple observation, that the vertices, which are close to each other according to graph distances, should also be drawn closer.

The algorithm 2 provides a simple heuristic approach for the k-clustering problem. In the new coarser graph the vertices in one cluster will be merged together in a single vertex - corresponding cluster center.

Algorithm 2 K-Centers$(G(V, E), k)$ [**?**]

1: $S = \{v\}$ for some arbitrary $v \in V$
2: **for** every $i = 2$ to k **do**
3: find u such that $min_{s \in S} d_{us} > min_{s \in S} d_{ws}\ \forall w \in V$
4: $S = S \cup \{u\}$
5: **end for**
6: **return** S

For local layout beatification the algorithm of the Kamada and Kawai (see section 2) with two small changes was selected. First, the algorithm is running predefined number of iterations without testing the condition in row 5, see algorithm 1. Second change is, that we consider only r-neighbourhood of each vertex, where r-neighbourhood of a vertex v is defined as $N^r(v) = \{u \in V | 0 \leq d_{uv} < r\}$. That means, that formula (1) in the modified version has the form:

$$E(p_1, \ldots, p_n) = \sum_{i \in V} \sum_{j \in N^r(i)} \frac{1}{2} k_{ij} \{|p_i - p_j| - l_{ij}\}^2 \tag{3}$$

The described variation of the Kamada and Kawai algorithm is summarized in algorithm 3.

Algorithm 3 LocalLayoutG($d_{V\times V}$, L, k, $Iterations$) [?]

for every $i = 1$ to $Iterations \cdot |V|$ **do**
    $v = argmax_u\ \Delta_u^k$
    compute δ_v^k by solving equations (2)
    $L(v) = L(v) + (\delta_v^k(x), \delta_v^k(y))$
end for

The complete algorithm of Harel and Koren is listed below:

Algorithm 4 LayoutG(V,E) [?]

Set constants Rad, $Iterations$, $Ratio$, $MinSize$
Compute the all-pairs shortest length $d_{V\times V}$
Set up random layout L
$k = MinSize$
while $k \leq |V|$ **do**
    $centers = \mathbf{K - Centers}(\mathbf{G(V,E)}, \mathbf{k})$
    $radius = \max_{v\in centers} \min_{u\in centers} d_{vu} \cdot Rad$
    LocalLayout$(d_{centers\times centers}, L(centers), radius, Iterations)$
    for every $v \in V$ **do**
        $L(v) = L(centers(v)) + rand$
    end for
    $k = k \cdot Ratio$
end while
return L

In the following section we represent the result of evaluation both algorithms in our implementation on the examples from corresponding papers.

4 Implementation

The both described algorithms were implemented with Python (version 2.7.6) and included in simple GUI for better visualisation proposes (see Fig. 1). The GUI was designed with QT Creator 3.2.2 and then bound with Python using PyQt version 4.11. The results of evaluation were obtained on two different laptops, but we mention by each table with performance results the hardware specification of the corresponding laptop.

In our code we used the following additional libraries : numpy (version 1.8.2) and scipy (version 0.13.3). Our implementation is unfortunately pure optimized, so that the better optimized

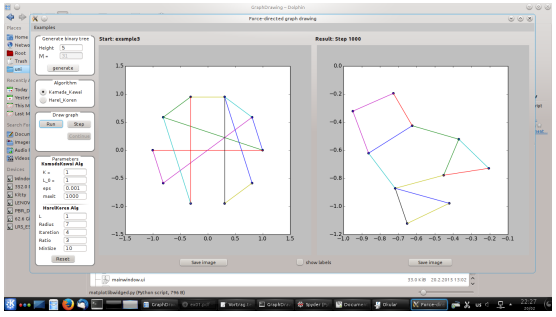

Figure 1: GUI

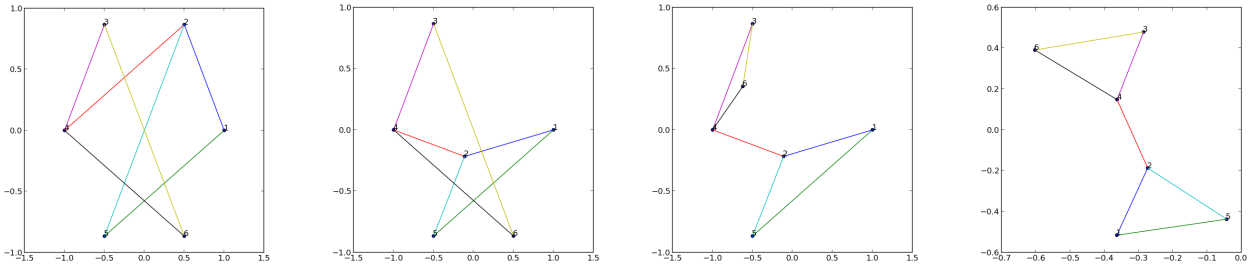

Figure 2: Drawing of a small graph in 4 different states, compare Fig.2 in [?]

version will probably perform better. But we did some optimization, because the initial implementation contained a lot of loops, which are rather slow in Python. So we used a vectorization to get rid of loops in the realisation of described algorithms.

It is also worth to mention that calculation of pairwise distance between all vertices of a given graph is extremely time consuming. In our implementation we first followed the idea in [?] and used Floyd's algorithm for shortest paths [?]. Since this has complexity $O(|V|^3)$ we also implemented a version of Dijkstras algorithm. For graphs with more than 1000 vertices we decided to use a heavily optimized version of Dijkstras algorithm from the scipy library. In the both papers [?] and [?] this problem did not arise, because the first one considers the graphs with fewer vertices and in the second an additional library for efficient algorithm on graphs was used.

5 Evaluation results

We initialize $p_1, \ldots, p_n$ uniformly on a circle with radius L_0. That means viewed as complex variables we can write

$$p_m = L_0 \times e^{2pi*i*m/n}$$

When we load a graph from a file it is also possible to start with a custom initialization of the p_m.

We tested both algorithm with different graphs from [?] and [?]. In figure 2 we show the initial state of a graph, the first two steps[1] and the state the graph converges to after 49 steps. In the first step the vertex v_2 gets moved to the middle. In the second step vertex v_6 gets moved. After this we already have a graph without edge crossing. In the following steps the movements are smaller and lead to a symmetric graph, where all the edges have the same length.

[1] we refer to one iteration of the outer loop as one step

Algorithm:		Kamda Kawai		Multi Scale	
Graph	$\|V\|$	Time	Iterations	Time	Iterations
Small Graph	6	0.169s	47	-	-
Pyramid Graph	10	0.370s	45	0.269s	1
Non Symmetric Graph	10	FC	FC	0.279s	1
Binary Graph(3)	7	0.136s	30	-	-
Binary Graph (4)	15	1.97s	131	0.373s	1
Binary Graph (5)	31	17.745s	323	7.021915s	2
	a	a	a		

Our results with KKA can be seen in the following table. Run time on an Intel Core i3-2310M CPU @ 2.10GHz $\times$ 4 . Parameter $K = 1,\ L_0 = 1,\ \epsilon = 0.001$. The Non Symmetric graph reaches the maximum number of iterations (FC failed to converge). But the drawing can be seen as good. For these small graphs, the multi scale algorithm is very fast, but it also doesn't reach acceptable drawings for our standard parametrisation.

6 Conclusions