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 [6]. In the first section we consider the approach by T. Kamada and S. Kawai [5] from 1988 and in the second section the multi-scale algorithm by D. Harel and Y. Koren [3], 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 [5]. 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

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

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 [5]

```
1: Compute the matrices d, l and k
2: initialize $p_1, \dots, p_n$
3: while ($\max \Delta_i > \epsilon$) do
4:     $m = argmax \Delta_i$
5:     while $\Delta_m > \epsilon$ do
6:         compute $\delta x$ and $\delta y$
7:         iterate $x_m = x_m + \delta x$, $y_m = y_m + \delta y$
8:         compute $\Delta_m$
9:     end while
10:    compute $\Delta_i$ for $i = 1, \dots, n$
11: 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 [3] (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 [2] and suggested new faster multi-scale algorithm.

The idea of the their algorithm is very simple: instead of considering the whole graph G at ones one builds an approximation sequence of coarse graphs $G^{k_1}, G^{k_2}, \dots, G^{k_l}$, $k_1 < k_2 < \dots < k_l = |V|$ (*multi-scale representation of* G), and applies one of the simple force-directed drawing algorithm to draw those graphs nicely. Beautification of each scale provides a locally nice layout of G and the sequence of all locally nice layouts approximates the nice layout of G. We refer to the paper [3] 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($G(V, E), k$) [3]

```
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 line 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:

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

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

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

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

The complete algorithm of Harel and Koren is listed below:

Algorithm 4 LayoutG(V,E) [3]

```
1: Set constants $Rad$, $Iterations$, $Ratio$, $MinSize$
2: Compute the all-pairs shortest length $d_{V\times V}$
3: Set up random layout L
4: $k = MinSize$
5: while $k \le |V|$ do
6:     $centers = \mathbf{K - Centers(G(V, E), k)}$
7:     $radius = \max_{v\in centers} \min_{u\in centers} d_{vu} \cdot Rad$
8:     $\mathbf{LocalLayout}(d_{centers\times centers}, L(centers), radius, Iterations)$
9:     for every $v \in V$ do
10:        $L(v) = L(centers(v)) + rand$
11:    end for
12:    $k = k \cdot Ratio$
13: end while
14: return L
```

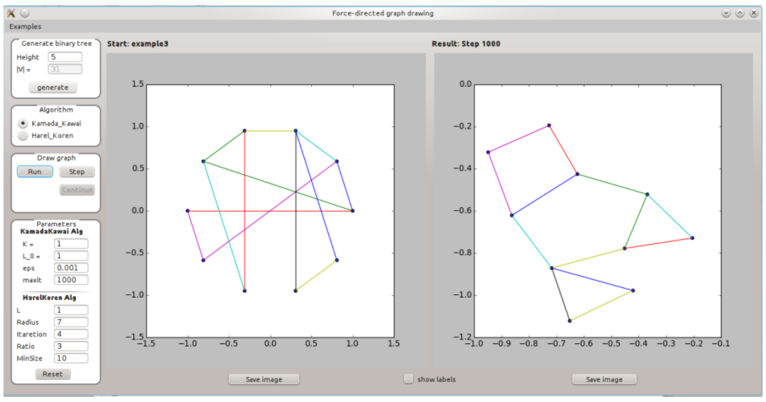
Notice a random noise ($(0, 0) < rand < (1, 1)$) added at line 10 to coordinates of vertices. In the following section we represent the result of evaluation both algorithms in our implementation on the examples from corresponding papers.

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 poorly optimized, so that the better optimized 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 mentioning that the calculation of pairwise distance between all vertices of a given graph is extremely time consuming. In our implementation we first followed the idea in [5] and used Floyd's algorithm for shortest paths [1]. 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 both papers [5] and [3] this problem did not arise, because the first one only considers graphs with fewer vertices and in the second paper an additional library for efficient algorithms on graphs was used.


Generate binary tree
Height 5
|V| = 31

Algorithm
[✓ Kamada_Kawai] Harel_Koren

Parameters
KamadaKawai Alg
K = 1
L_0 = 1
eps 0.001
maxit 1000

HarelKoren Alg
L 1
Radius 7
Itaretion 4
Ratio 3
MinSize 10

Figure 1: GUI

5 Evaluation results

As start layout we initialize vertices of the graph uniformly on a circle with radius L_0. That means viewed as complex variables we can write

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

For some examples we load the initial positions of vertices from corresponding input file (example of the 3elt graph from Scotch collection [7])

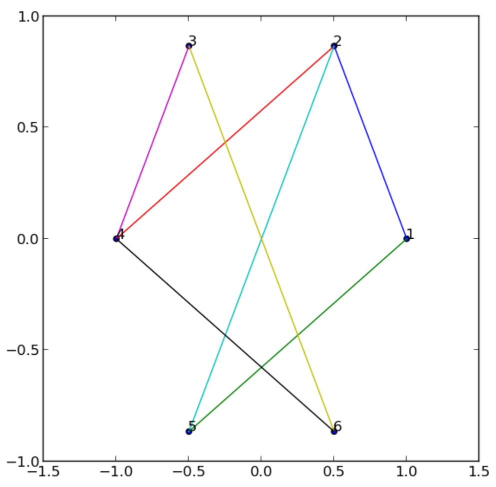
We tested both algorithm with different graphs from [5] and [3].

Algorithm of Kamada and Kawai

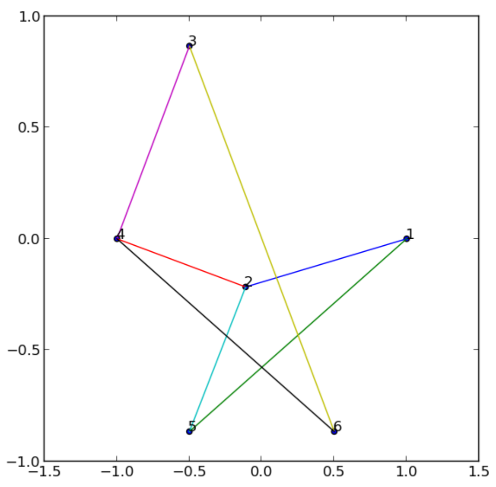
In Fig. 2 we show the initial state of a graph, the first two steps [1] and the state the graph converges to after 49 steps. In the first step the vertex v_2 gets moved to the middle. In the second step vertex v_6 gets moved. After this we already have a graph without edge crossing. In the following steps the movements are smaller and lead to a symmetric graph, where all the edges have the same length.
We also replicated the results from Fig.3 b and Fig. 5 in [5] (compare Fig. 3 and Fig. 4). It is interesting to see that the non symmetric graph in Fig 4 is not a minimum. The drawing of the graph rotates along its center. In Fig. 5 we plotted the result drawings of a binary tree achieved by both algorithms. The result achieved by KKA looks slightly better. We believe that for graphs with less than 100 vertices KKA is generally preferable.

Our results with KKA can be seen in table 1. Run time on an Intel Core i3-2310M CPU @ 2.10GHz $\times$ 4 . Parameters were set to $K = 1$, $L_0 = 1$, $\epsilon = 0.001$ for KKA and $MinSize = 10$, $Ratio = 3$, $Iterations = 4$, $Rad = 7$ for HKA. 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 doesn't reach nice drawings for our standard parametrisation. In these cases we omitted the results. The table also illustrates the importance of vectorisation when working in python.

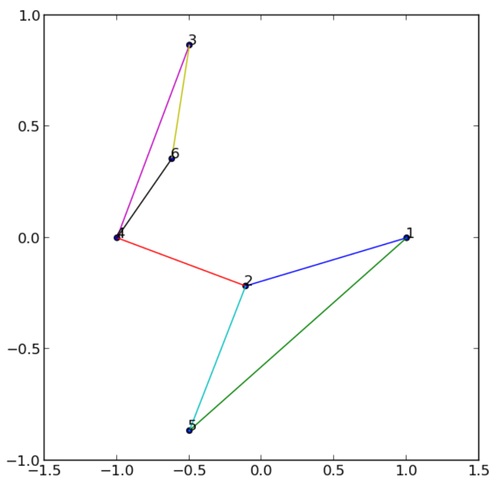
[1]we refer to one iteration of the outer loop as one step



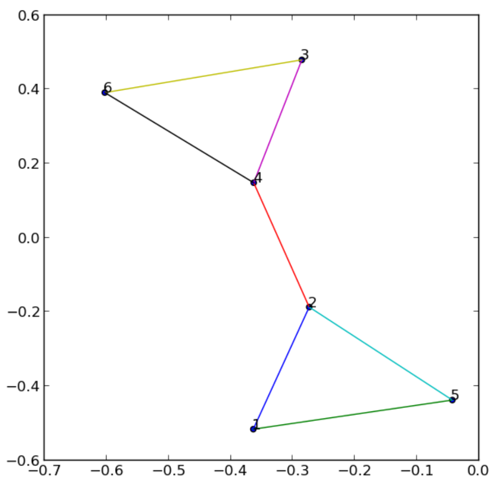
(a) Initial State



(b) Step 1



(c) Step 2

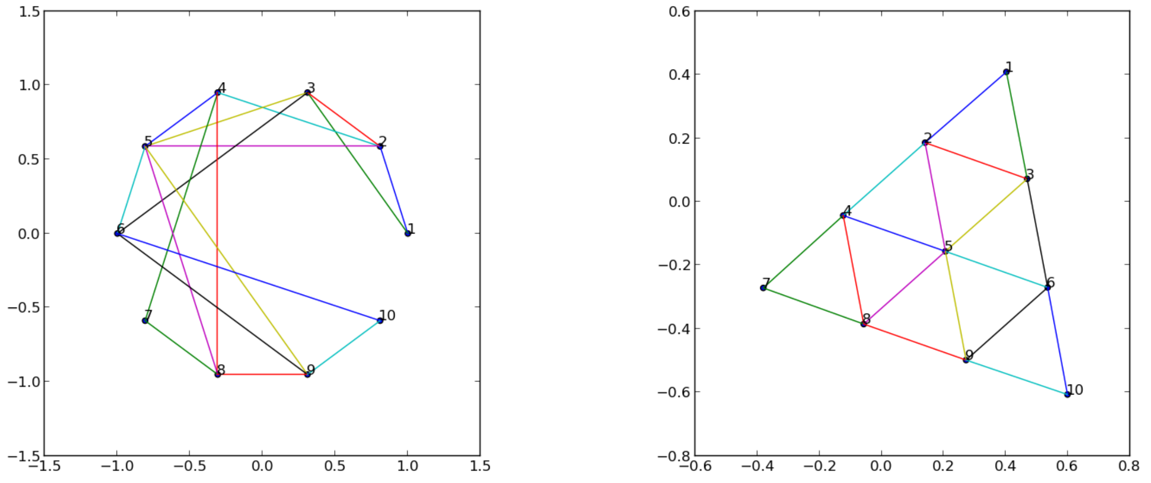


(d) Step 49

Figure 2: Drawing of a small graph in 4 different states, compare Fig.2 in [5]

Algorithm:		Kamada Kawai w. loops		K. K. optimized		Multi Scale	
Graph	$\|V\|$	Time	Iterations	Time	Iterations	Time	Iterations
Small Graph	6	0.169s	47	0.05s	53	-	-
Pyramid Graph	10	0.370s	45	0.05	46	0.021s	1
Non Symmetric Graph	10	FC	FC	FC	FC	0.024s	1
Binary Graph(3)	7	0.136s	30	0.038s	31	-	-
Binary Graph (4)	15	1.97s	131	0.11s	100	-	-
Binary Graph (5)	31	17.745s	323	0.50s	366	0.032s	1

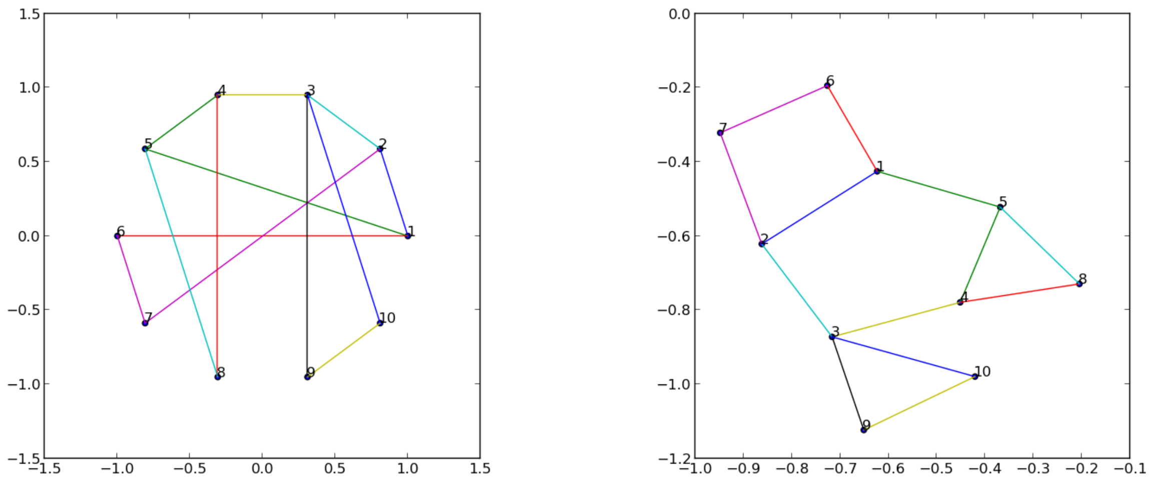
Table 1: Results for small graphs



(a) Initial State

(b) State after 46 steps

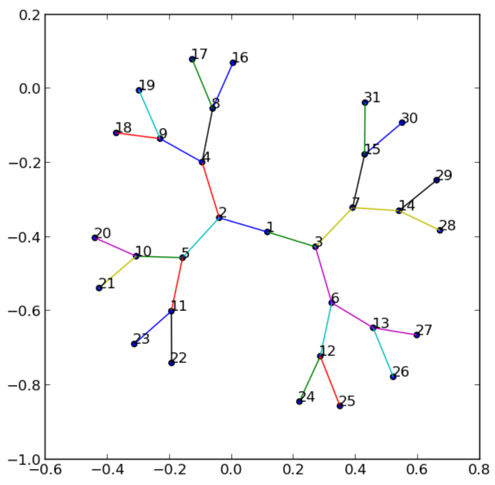
Figure 3: Pyramid graph (compare figure $3b$ in [5])



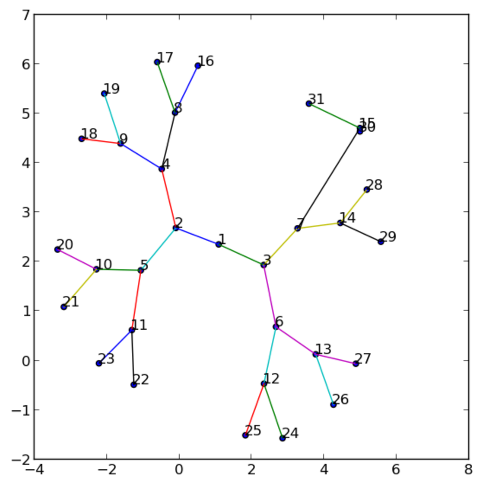
(a) Initial State

(b) State after $maxit = 1000$ steps

Figure 4: Non symmetric graph (compare figure 5 in [5])



(a) Result with KKA



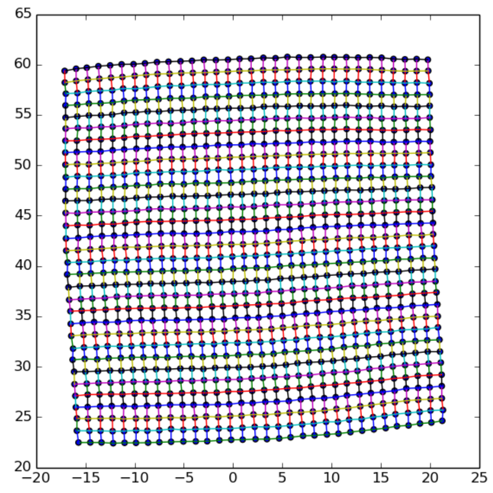
(b) Result with HKA

Figure 5: Full Binary tree of depth 5 for both algorithms

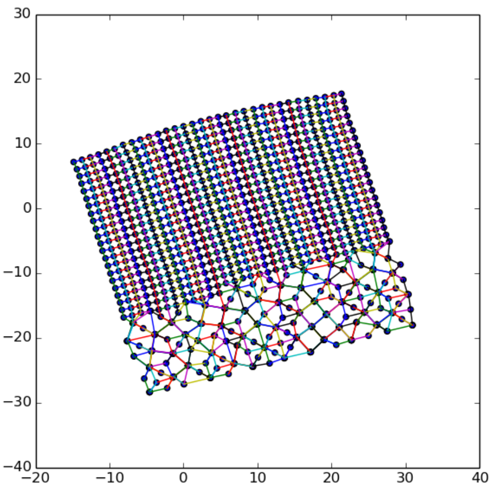
Algorithm of Harel and Koren

Compared with the Algorithm of Kamada and Kawai the algorithm of Harel and Koren includes 4 parameters: minimal size of the coarsest graph (*MinSize*), ratio between number of vertices in two consecutive levels (*Ratio*), number of iterations of local beautification (*Iterations*), size of the neighbourhood (*Rad*) (as a further parameter one can consider the desired length of graph edges (L), but the authors do not mention it specially). The authors claim, that for all examples in their paper they used the following parameters: $MinSize = 10$, $Ratio = 3$, $Iterations = 4$, $Rad = 7$ (for the complete binary tree *Rad* was set to 16 or 17) and achieved reasonable results.

Unfortunately, that was not always the case in our evaluation. So on figure 6 one can see that in case of square grid graphs some vertices stay clustered after the final iteration of the algorithm with the default settings. On the other hand, in case of complete binary tree default setting deliver better drawing results (compare 7).

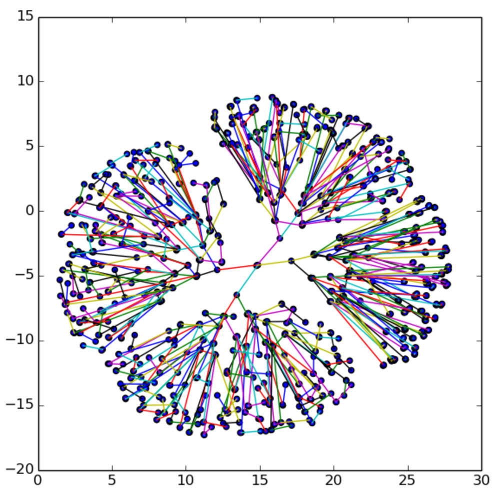


(a) $MinSize = 2$, $Ratio = 2$



(b) $MinSize = 10$, $Ratio = 3$

Figure 6: Square grid graph 32×32 (compare figure 1 in [3])


(a) $MinSize = 2$, $Ratio = 2$, $Rad = 18$

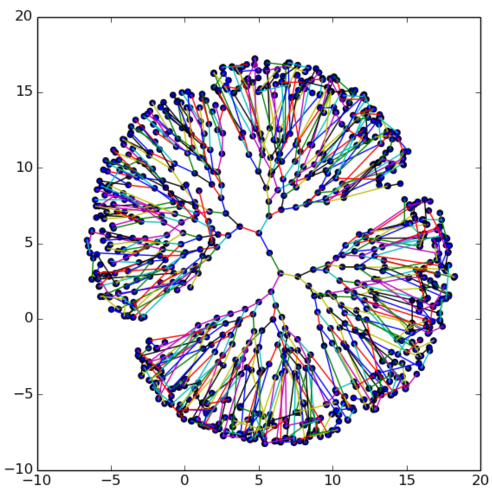
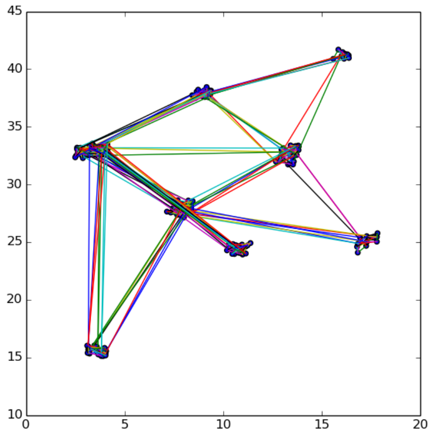
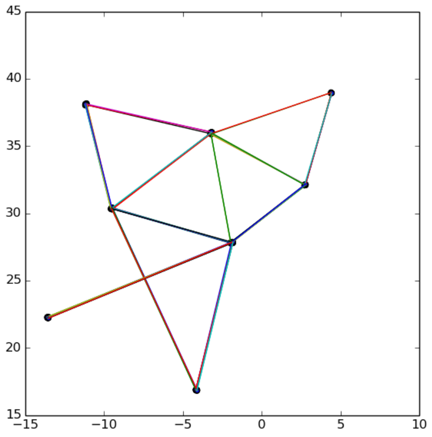

(b) $MinSize = 10$, $Ratio = 3$, $Rad = 18$

Figure 7: Complete binary tree with 1023 vertices (compare figure 6 in [3])

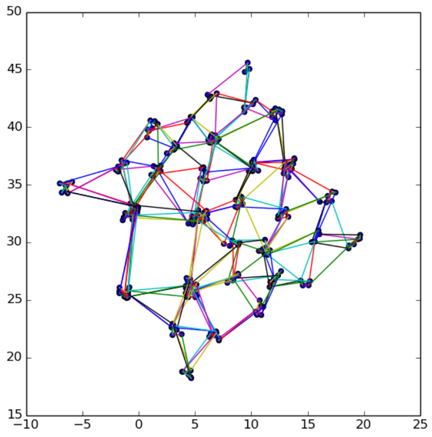
We also find out, that a random noise added to vertex coordinates (see line 10 in algorithm 4) has some impact on the end result. On the fig. 8 one can see that adding smaller noise (for example from $(0, 0)$ to $(0.1, 0.1)$) leads to better drawing result in our implementation.



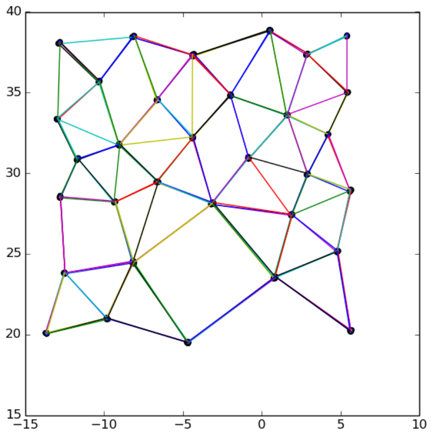
(a) Step 3



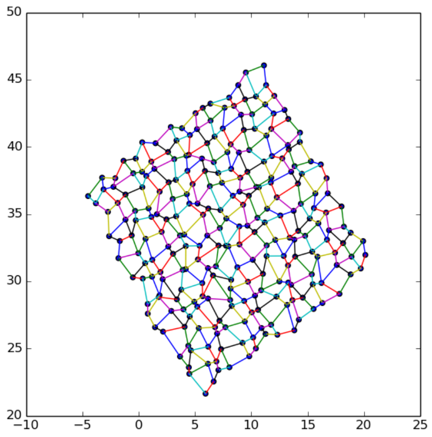
(b) Step 3



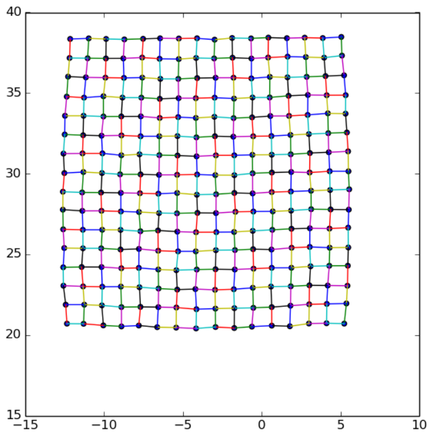
(c) Step 5



(d) Step 5



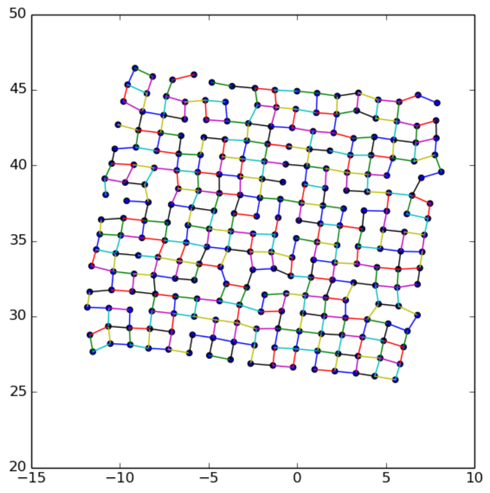
(e) Step 8



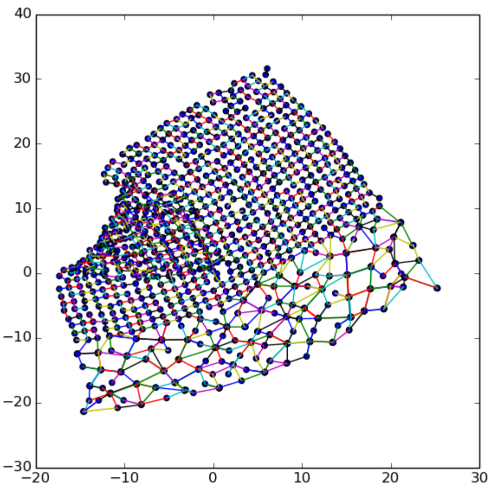
(f) Step 8

Figure 8: Influence of the random noise added to coordinates of vertices on the example of grid graph 16×16. Left column : $(0, 0) < rand < (1, 1)$, right column - $(0, 0) < rand < (0.1, 0.1)$

In figure **??** we show the result of applying the HKA to the sparse grid graphs (1/3 of edges were omitted). Unfortunately, the images do not look so spectacular as in original paper (see figure 4 in [3]).



(a) Sparse square grid graph 16×16 (256 vertices), $MinSize = 2$, $Ratio = 2$



(b) Sparse square grid graph 32×32 (1024 vertices), $MinSize = 10$, $Ratio = 3$

The performance of our implementation of HKA can be seen in table 2. Computations were made on an Intel Core i7-3520M CPU @ 2.90GHz.

Algorithm:		Kamada Kawai		Multi Scale		Parameters
Complete Binary Graph	255			0.626503s	3	$MinSize = 10$, $Ratio = 3$, $Rad = 18$
Complete Binary Graph	1023			1402.674150s	5	$MinSize = 10$, $Ratio = 3$,$Rad = 18$
Complete Binary Graph	1023			??????s	9	$MinSize = 2$, $Ratio = 2$, $Rad = 18$
Square Grid Graph	256			20.029714s	8	$MinSize = 2$, $Ratio = 2$
Square Grid Graph	256			0.615102s	3	$MinSize = 10$, $Ratio = 3$
Square Grid Graph	1024			3854.181407s	10	$MinSize = 2$, $Ratio = 2$
Square Grid Graph	1024			1403.437832s	5	$MinSize = 10$, $Ratio = 3$
Sparse Square Grid Graph	256			20.168194s	8	$MinSize = 2$, $Ratio = 2$
Sparse Square Grid Graph	1024			1439.099359s	5	$MinSize = 10$, $Ratio = 3$

Table 2: Results for large graphs

6 Conclusions

We successfully implemented the algorithms described in [5] and [3] and were able to replicate most of their results. We found out that KKA is a good choice for drawing small graphs but takes too long for a high number of vertices. HKA is much better suited for handling larger graphs. But it also has the problem that we need to compute and store the complete distance matrix d, which is expensive. Hence, we encountered difficulties with the graphs that had more than 1024 vertices. If we were to keep working on the graph drawing problem we see two possibilities. We could try to further optimize our implementation of HKA. However we might need to port it in a language like C++ to achieve the speed described in [3]. The more interesting thing to do would be to follow another idea by Harel and Koren and take a look at [4]. Here the authors propose a new approach where they first embed the graph in a high dimensional space and then project it into 2D using principal component analysis. This would lead us back to the theory we encountered in our Machine Learning lecture.

References

[1] R. W. Floyd, *Algorithm 97: shortest path*, Comm. SCM 5, 5 (1962), pp. 279–281.

[2] R. Hadany and D. Harel, *A multi-scale algorithm for drawing graphs nicely*, Discrete Apllied Mathematics, 113 (2001), pp. 3, 21.

[3] D. Harel and Y. Koren, *A fast multi-scale method for drawing large graphs*, Journal of Graph Algorithms and Applications, 6 (2002), p. 179.202.

[4] D. Harel and Y. Koren, *Graph drawing by high dimensional embedding*, Journal of Graph Algorithms and Applications, 8 (2004), pp. 195–214.

[5] T. Kamada and S. Kawai, *An algorithm for drawing general undirected graphs*, Information Processing Letters, 31 (1989), pp. 7–15.

[6] S. G. Kobouro, *Force-directed drawing algorithms*, ch. 12.

[7] J. Petit, *Minimum linear arrangement problem: instances, codes and results.*