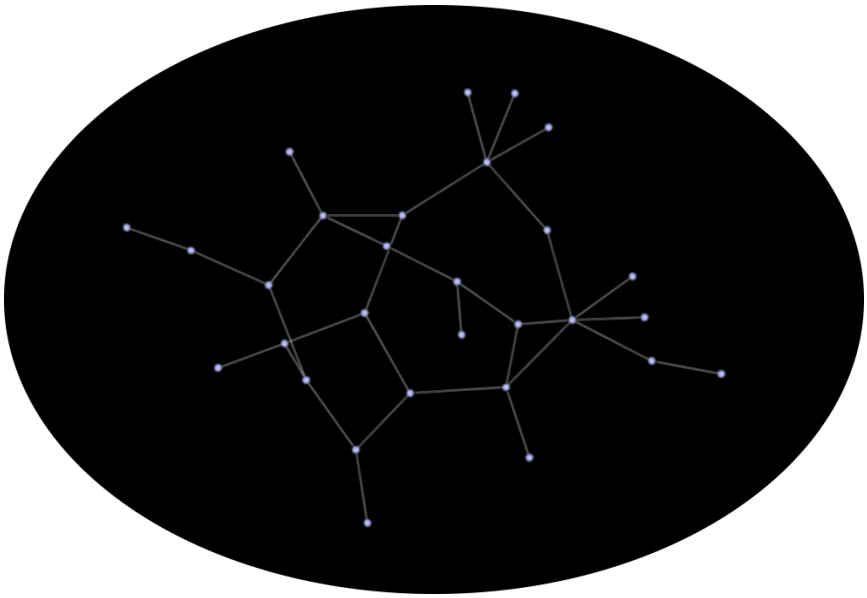
Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and
Networks Laboratory
Communication Systems Group

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Force-based Visualization of Peer-to-peer Nodes

Semester Thesis



SA-2007-01

WS 2006/07

Elias Bürli

Assistant: Marcel Baur
Professor: Prof. Bernhard Plattner

Abstract

In this thesis, we examine several methods to automatically generate graphs. The graphs are used to display the topology of a peer-to-peer network. Evaluation criteria are stated to compare the results of different drawing algorithms.
The chosen algorithm is a variation of the force-based algorithms. It includes sophisticated heuristics to improve convergence behavior.
The implemented solution is tested with both generated data and actual data from the running network. The resulting graphs are compared to the stated evaluation criteria.

Zusammenfassung

In dieser Semesterarbeit werden mehrere Methoden zur automatischen Generierung von Graphen untersucht. Die Graphen stellen die Topologie eines Peer-to-peer Netzwerkes dar. Evaluationskriterien werden aufgestellt, um verschiedene Zeichenalgorithmen miteinander zu vergleichen.
Ein kräftebasiertes Verfahren wird implementiert, das mit heuristischen Methoden erweitert wurde, um das Konvergenzverhalten zu verbessern.
Die implementierte Lösung wird sowohl mit generierten Daten als auch mit richtigen Daten des aktiven Netzwerks getestet. Die resultierenden Graphen werden anhand der zuvor aufgestellten Evaluationskriterien bewertet.

Table of Contents

1 Introduction and problem statement

Celeste [1] is a shared, highly reliable and distributed data storage system. It implements an automatically managed and secure storage for mutable data on top of an untrusted peer-to-peer platform for multiple users.

Along with Celeste, a graphical application named *Celeste Visualizer* was developed, which is used to display peer-to-peer nodes and their actions by registering with the overlay nodes as a listener.

The goal of the thesis was to clean up and extend the existing *Celeste Visualizer* code with robust and efficient algorithms to display nodes and their relationships.

The main task of the thesis was to identify and evaluate algorithms for expedient display of the Celeste network. A suitable algorithm (possibly several) was to be implemented and tested in the real Celeste network. Along with the display algorithm some additional functionality for navigating within the diagram as well as displaying additional information of single nodes was to be added to the application.

The challenge lies in the dynamic nature of the Celeste network. Nodes may appear and disappear during runtime, which results in a somewhat diffuse cloud of nodes, that ought to be presented in a comprehensible form. A promising approach is the use of graph drawing algorithms to automatically generate a diagram out of the raw data.

Section 2 covers possible solutions and evaluations, while the actual implementation is described in section 3. Finally section 4 contains the evaluation of our work, with the conclusions and possible future work listed in section 5.

2 Background and related work

In this section we review existing work in the area of automated graph drawing. Evaluation criteria are stated with which different algorithms are compared.

2.1 Overview

Since the Celeste network consists of nodes and corresponding edges, it can be represented as a graph [12]. This allows us to apply graph drawing methods to the given Celeste network data.

With the rise of more powerful personal computers, the automatic generation of graphs has become a more active research field within the last few decades. Several different approaches and algorithms for the problem have been stated in the past years, mostly differing in the criteria for a nice graph, computation time and the types of graphs for which they are best suited.

The first major papers were published during the early 70's [2], and the first “Graph

Drawing" conference was held in 1992. Since then several new and advanced algorithms have been developed, with the goal of automating the process of drawing a nice graph.

Yet the problem is inherently ill-defined. What exactly does *nice* mean? We seek an algorithm that shows off the structure of the graph so the viewer can best understand it. We also seek a drawing that looks aesthetically pleasing. Unfortunately, these are *soft* criteria for which it is impossible to design an optimization algorithm. Indeed, it is possible to come up with two or more radically different drawings of certain graphs and have each be most appropriate in certain contexts.

To get at least some rough formal criteria we have to consult perception psychology. Studies in this field [3] reveal that we can define some common rules for an aesthetically pleasing and comprehensible graph, that will be examined in the next section.

2.2 Evaluation criteria

The possible classes of algorithms were examined regarding their suitability for *Celeste Visualizer*. The optimal goal was to achieve an aesthetically pleasing layout within "interactive" time, meaning less than a second. Of course computation time will always depend on the hardware the program is run on, thus no "hard" limit can be stated.

As for aesthetic criteria, the widely accepted rules to draw a comprehensible graph can be summed up as follows:

1. Even distribution of the vertices in the available space
2. Uniform edge length
3. Minimized edge crossings
4. Reflection of inherent symmetry
5. Avoidance of sharp angles between edges

Fig.1 shows two different drawings of the same graph. *a)* is a random layout, while *b)* is a hand-drawn layout following above stated rules. The difference in comprehensibility is striking.

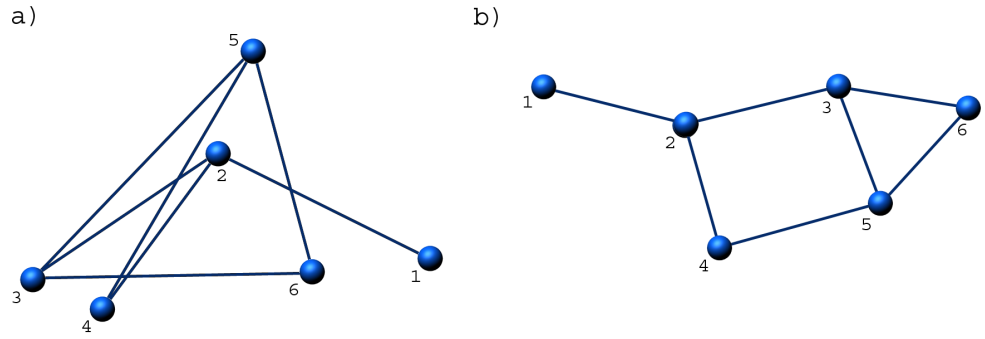


Fig. 1. Two drawings of the same graph, a) ignoring and b) following the stated criteria

Several different classes of algorithms exist automatically generate graphs according to the aesthetic criteria, although no one can satisfy all of them optimally. Therefore, a trade-off has to be made to chose an algorithm that satisfies most criteria according to our needs.

2.3 Planarization algorithms

By definition a planar graph is one which can be drawn on a sheet of paper without any crossing edges. Ever since an efficient planarity test for graphs had been published, algorithms for the planarization of graphs have been implemented [4, 5, 6]. Many of them have some restrictions concerning a maximum degree of the vertices or a minimum n-connectivity of the graph, which limits their practicability since nothing is known about the topology of the Celeste network before runtime.

If such an algorithm was to be applied to arbitrary graphs, one has to fulfill the restrictions by removing the minimum number of edges until the graph can be drawn planar and reinstate the removed edges afterwards. This method guarantees minimal crossing on the cost of the other criteria, the resulting graphs suffer from uneven use of the available space, unnecessary sharp angles between edges and the edges themselves tend to get very long and need multiple bends as the graph complexity increases. Fig. 2 (from [10]) shows a comparison of two drawing of the same graph, *a)* before and *b)* after a planarization algorithm was applied.

There exist more advanced versions of planarization algorithms [10] that produce better results, but are significantly more computing intensive. In worst case scenarios this can mean computing times of several minutes for less than 100 vertices as opposed to a few seconds for the simpler algorithms. This is not a concern for the main applications of these advanced planarization methods, mainly VLSI routing problems, but makes them too slow for our purposes.

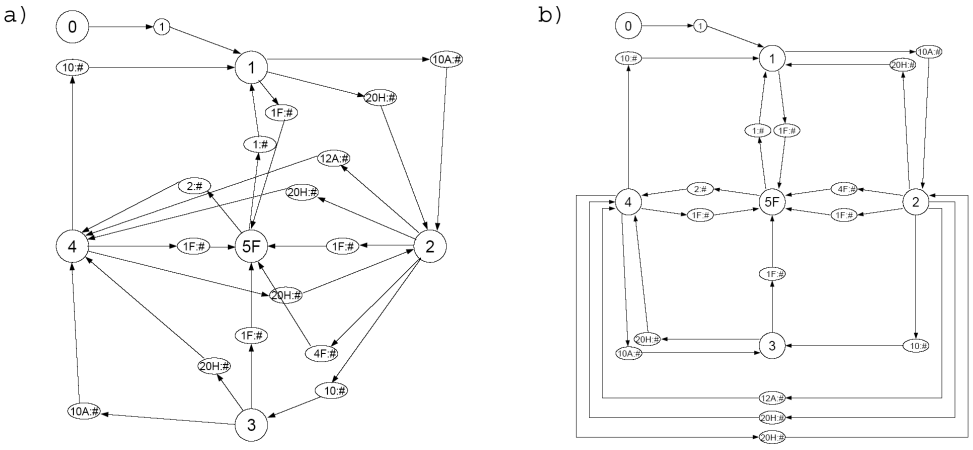


Fig. 2. Two drawings of the same graph, before (a) and after (b) a planarization algorithm was applied.

2.4 Hierarchical algorithms

Another class of drawing methods are the hierarchical algorithms [7], whose goal is to preserve any given hierarchy present in the structure represented by graph, which makes them best suited for directed graphs. To achieve this, all vertices are assigned to a horizontal layer according to their hierarchical level as the first step. In a second step the vertices within a layer are arranged in a way that minimizes the edge crossings between the layers. This second step can be computing intensive and is therefore often solved by heuristic methods.

Since the Celeste network is not inheriting a strict hierarchy and the edges are usually not directed, these algorithms are of little practical use for it.

Furthermore these algorithms are optimized for minimal edge crossings, ignoring several of the other aesthetic criteria. This results in reduced comprehensibility when applied to more complex and arbitrary graphs. Fig. 3 shows an example of a hierarchical graph layout, taken from [8]. Clearly visible are the horizontal hierarchy levels, while the long edges are tightly packed together to achieve minimal crossings.

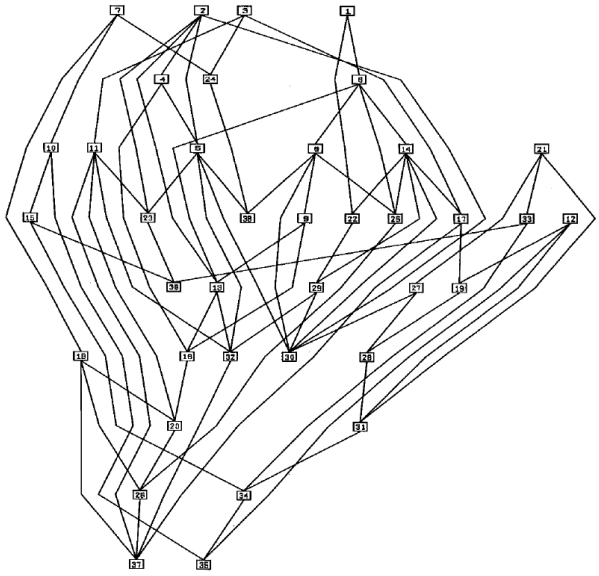


Fig. 3. Example of a hierarchical graph layout

2.5 Force-based algorithms

Especially suited for sparse graphs are force-based algorithms [8,9]. Originally introduced by Eades [11], they are sometimes also referred to as spring embedder methods. While many modified versions of the algorithm were developed, they all share a common underlying physical model. Vertices are treated as charged particles exercising repelling forces on each other. The edges on the other hand are modeled as linear springs, acting as attractive forces between vertices that are connected by an edge.

Executing the algorithm means simulating this physical model in iterative steps until the forces arrive in an equilibrium, which is equivalent to a minimum energy state. While this is a heuristic approach to the problem, as opposed to the analytical method of planarization algorithms, the resulting graphs converge toward a state with evenly distributed vertices, uniform edge lengths and maximized angles between edges. Fig. 4 shows a typical example of a graph layout obtained from a force-based algorithm (from [9]). Due to the physical properties of the model, inherited symmetries are also bound to show up.

The drawback of this approach is that minimized crossings are not guaranteed but only probable. Computing time is that of an n-body problem, but it should be noted that several of the newer versions of this algorithm introduce methods to speed up the iteration. This led to the decision that a force-based approach was the most promising and therefore chosen for implementation.

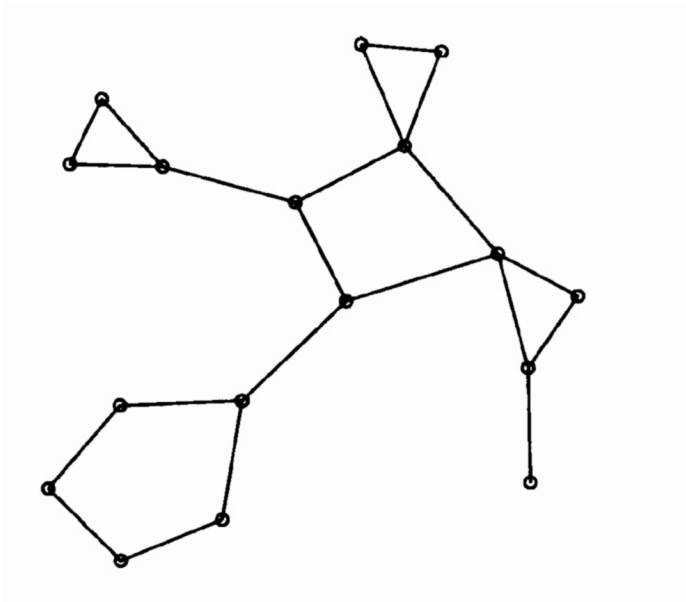


Fig. 4. Typical example of a force-based layout

3 Implemented solution

The chosen algorithm, described by Frick, Ludwig and Mehldau in [13], is a variation of the force-based method originally described by Eades [11]. It extends the previous versions with more sophisticated heuristics to improve convergence.

3.1 Concept

The *GEM* (short for *graph embedder*) algorithm keeps the underlying physical model from Eades' method with the vertices applying repelling forces on each other, while the edges between them act a springs trying to maintain a desired length.

Starting with an random initial placement of vertices, the original algorithm iterates the system in discrete time steps until a fixed number of steps was taken. This approach has the obvious drawback that the system may not have converged by then or it may have wasted unnecessary iterations if the fixed number of steps was chosen too large. Since the algorithm performs a *gradient descent*, this can occur if the system gets trapped in a local minimum.

To overcome this problem, a technique from statistical mechanics called *simulated annealing* was introduced, which allows the system to change into a state of higher energy. After each computation step the state change is examined for its change in the *energy landscape,* referring to the net energy of the system. Any downhill move is accepted, while uphill moves are accepted with a probability depending on the current *temperature*. This allows the system for arbitrary movements at the beginning of the iteration, when the temperature is relatively high. But as the iteration progresses the system temperature decreases and thus the probability of choosing a next state with more energy approaches zero. Such a cooling schedule was introduced by Fruchterman and Reingold [9].

What sets the *GEM* algorithm apart from the previous versions is the introduction of a more sophisticated cooling schedule to improve convergence, although it is not a cooling schedule in the strict sense. Rather, the algorithm adapts to the data locally and does not rely on global cooling as assumed by a schedule. For each vertex a *local temperature* is defined that gets adjusted according to the previous temperature and the likelihood that the vertex is oscillating or part of a rotating subgraph. The detection of oscillations and rotations is another unique feature of the *GEM* algorithm.

The local temperature is risen if it is determined that a vertex is probably not close to its final stable position. As a measure of the current stability of the graph the *global temperature* is defined as the average of the local temperatures over all vertices.

To further accelerate the convergence of *GEM*, a gravitational force is introduced, pulling all vertices towards the *barycenter* of the vertex cluster. It also helps to keep disconnected graphs and loosely connected components together.

Figure 5 shows schematically all the forces acting on one particular vertice. Shown are the current vertice *v*, the repelling forces from other nodes (blue vectors), the attractive forces of adjacent vertices (red vectors) and the gravitational force (green vector) to the barycenter B.

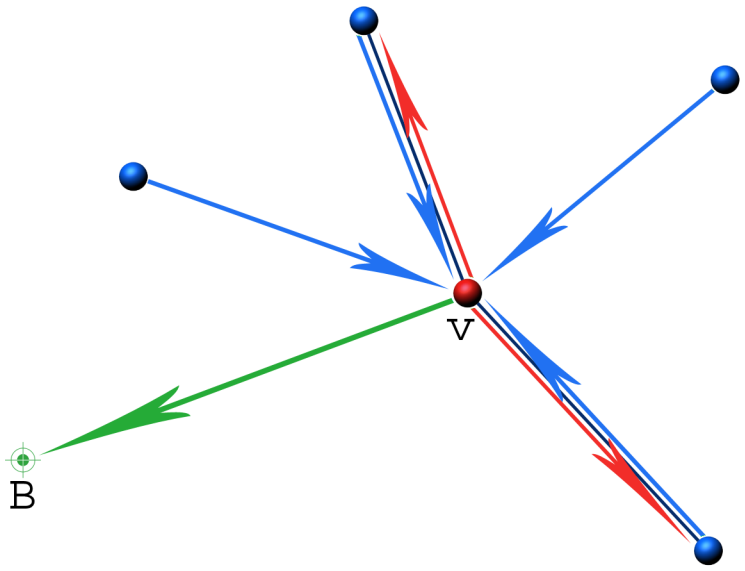


Fig.5. Total forces acting on a single vertice v, B being the barycenter

3.2 Implementation

Within the *Celeste Visualizer* application separate hashtables are used to store individual nodes and edges. Besides various data used for other functions of the application, each node stores its position, local temperature, impulse and skew.

When the force-based node arrangement is selected, *Celeste Visualizer* calls the main function of the *GEM* algorithm, which is given in pseudo code in Fig. 6.

The *GEM* algorithm consists of two stages, an initialization stage and an iteration stage. In the initialization stage a random initial position, a zero impulse vector and an initial temperature is assigned to each vertice.

The iteration stage sequentially updates vertex positions and local temperatures until the global temperature is lower than a desired minimal value or the time allowance has expired.

```
procedure GEM is {
--Input:
--  graph G = (V, E) where
--  V = set of all vertices, each vertice containing
--          ξ  current position
--          p  last impulse
--          t  local temperature
--          d  skew gauge
--  I_max   maximum number of iteration steps
--  T_max   upper bound on local temperature
--  T_min   desired minimal temperature
--Output:
--  for each  v∈V  , a new position is computed

for (  all v∈V  ) {
          initialize(v)
}

while (T_global > T_min && number of iterations <  I_max) {
          choose next vertex v to update
          compute impulse of v
          update position and temperature of v
} }
```

Fig.6. Main loop of the GEM algorithm

The sub functions for impulse computation and temperature adjustment are given in Fig. 7 and Fig. 8 respectively.

```
--Input:
--  v      vertex to be updated
--  c      barycenter of G
--  Φ      function growing with deg(v)  [1 + deg(v)/2]
--Output:
--  p      current impulse of v
--Constants:
--  E_des  desired edge length  [50]
--  γ      gravitational constant  [0.18]

p := (c - v.ξ) * γ * Φ(v)
-- random disturbance
δ := small random vector
p = p + eta
for ( all u∈V ) {
        --repulsive forces to all other vertices
        Δ := v.ξ – u.ξ
        if (Δ ≠ 0) then p := p + Δ * E_des^2/|Δ|^2
}

for ( all u,v∈E ) {
        --attractive force between u and v
        Δ := v.ξ – u.ξ
        p := p - Δ * |Δ|^2/( E_des^2 * Φ(v))
}
```

Fig. 7. Impulse computation of the GEM algorithm

The impulse is *GEM*'s way of keeping track of the last movement of each vertex. It is governed by several global constants ,a desired edge length and a gravitational constant factor GAMMA determining how a strongly a vertex is driven towards the barycenter. The resulting force on a given vertex is the superposition of the repelling forces of the other nodes, the attractive forces to adjacent nodes, the gravitational force and a small random disturbance.

The function Φ is a scaling factor giving vertices with many edges more inertia. This improves the layout quality in some cases by keeping them close to the barycenter.

```
--Input:
--  v       vertex to be updated
--  p       current impulse of v
--Output:  v with updated ξ, t, d, p
--Constants:
--  Tmax    maximum temperature [256]
--  αo      opening angle for oscillation detection [π/2]
--  αr      opening angle for rotation detection [π/3]
--  σo      sensitivity towards oscillation [1/3]
--  σr      sensitivity towards rotation [|V|/2]

if (p ≠ 0) then {
        p := v.t * p/|p|  --scale with current temperature
        v.ξ := v.ξ  + p
}
if (v.p ≠ 0) then {
        β := ∠(p, v.p)
        if ( β ≥ 1/2*(π - αr) &&   β ≤ (1/2*(π – αr)) then {
                -- rotation detected
                v.d := v.d + σr * sin(β)
        }
        if ( β >  π - 1/2* αo) {
                -- oscillation detected
                v.t := v.t *  σo * cos(β)
        }
        v.t := v.t * (1 - |v.d|)
        v.t := min(v.t, Tmax)
        v.p := p
```

Fig. 8. Temperature update algorithm

After the impulse for the current vertex *v* is calculated, its position is updated. If *v*'s impulse was non-negligible we update its internal data structures.

A new local temperature for *v* is computed based on the last temperature, the last and current movement and the skew gauge *d*. The skew is an indicator for the likeliness of *v* oscillating or being part of a rotation. Rotations can occur when the final layout has been found, but the temperature is still too high for the graph to come to rest. Under rare circumstances a rotating graph never converges, so cooling down is an appropriate reaction whenever significant rotations are detected.

Oscillations on the other hand are suspected when the last and current impulse vectors point in opposite directions. In that case *GEM* assumes that the vertex has just passed its equilibrium position and lowers the temperature according to a sensitivity factor σ. Subsequent oscillations will therefore finally freeze the vertex.

The detection of rotations and oscillations require knowledge of β, the angle between the current and previous impulse vectors. For an in-depth discussion of the detection rules we refer to [13].

4 Evaluation

4.2 Evaluation with random graphs

The results of applying the force-based algorithm to randomly generated graphs can be seen in Fig. 9. Graph *a)* on the upper left shows a small graph with 16 vertices and 17 edges. To judge its quality we compare it with the aesthetic criteria from section 3.2.

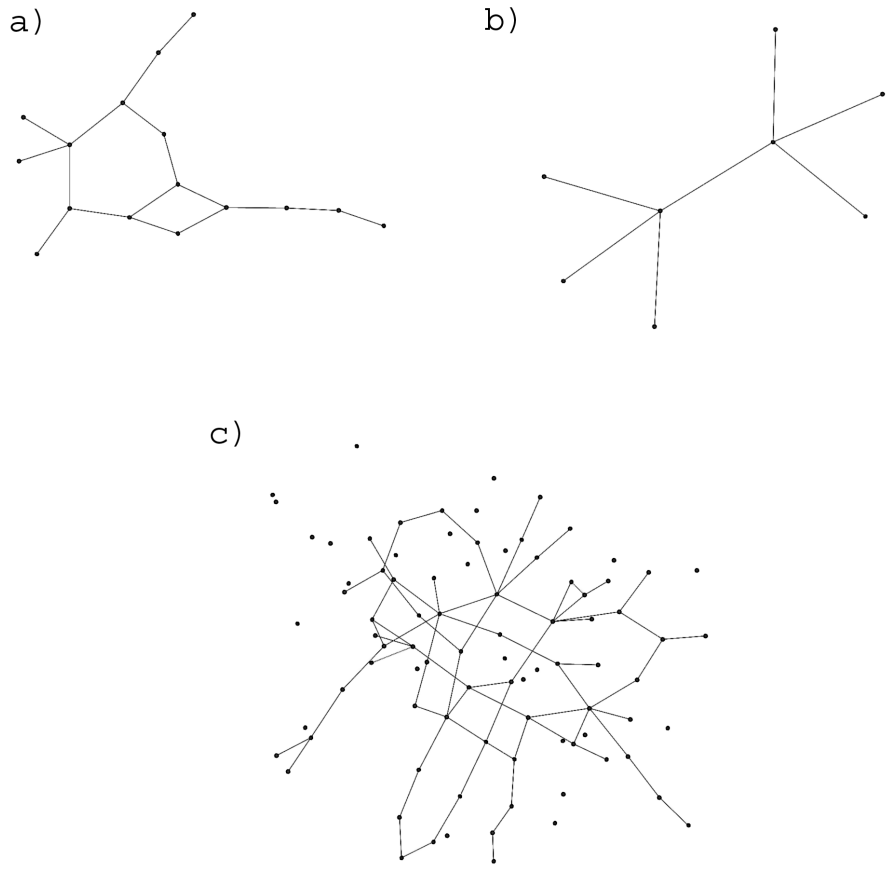


Fig. 9. Examples of a) small, b) symmetric and c) complex resulting graphs

The graph makes good use of the available space and the edge lengths are quite uniform across the graph. Also the angles between edges tend to become as large as possible as a result of the repellent forces between the vertices and the uniform edge lengths. If the graph is inheriting a symmetry, it shows up clearly in the resulting layout, as seen in Fig. 9b on the upper right.

Where the algorithm falls short is the desire for minimal edge crossings. That's not surprising since there's no mechanism to prevent unnecessary crossings, although for simple planar graphs the result is often free of edge crossing. But as the complexity of the graph increases, more unnecessary crossings and sharp angles between edges start to appear. This is evident in graph *c)* with 100 vertices and 80 edges. Uniform edge lengths and even vertice distribution on the other hand don't decrease with growing graph complexity. This allows the viewer to get an overview of the whole Celeste network even if the graph gets very complex. Vertices with the highest degree usually end up in the middle of the resulting graph, while the lower degree vertices drift to the outer regions.

Table 1 shows average computing times for increasingly complex graphs with vertices |V| and edges |E|. While the computing time cannot be expressed explicitly, heuristic estimations state that approximately |V| rounds are needed. Since each round consists of |V| iterations and each iteration considers |V| vertices, the time complexity is of order $O(|V^3|)$.

The tests were run on a 1.86 GHz X86 machine, which allowed near-interactive response times up to approximately 100 vertices.

|V|	**|E|**	**computing time [s]**
25	20	0.06
50	40	0.28
75	60	0.59
100	80	1.37
150	120	3.72
200	160	7.54
250	200	10.89
300	240	16.44
500	400	65.28

Table 1. Average computing times

4.2 Evaluation with Celeste on PlanetLab

After testing the algorithm with randomly generated graphs, further test runs were made running *Celeste* on PlanetLab[14]. PlanetLab is an open platform for developing planetary-scale services. It consists of several hundreds of PCs distributed over dozens of countries.

Fig. 10 shows a network of 82 Celeste nodes being displayed by *Celeste Visualizer*. In this example the overall network structure can be seen quite clearly and the edge crossings are not obstructing.

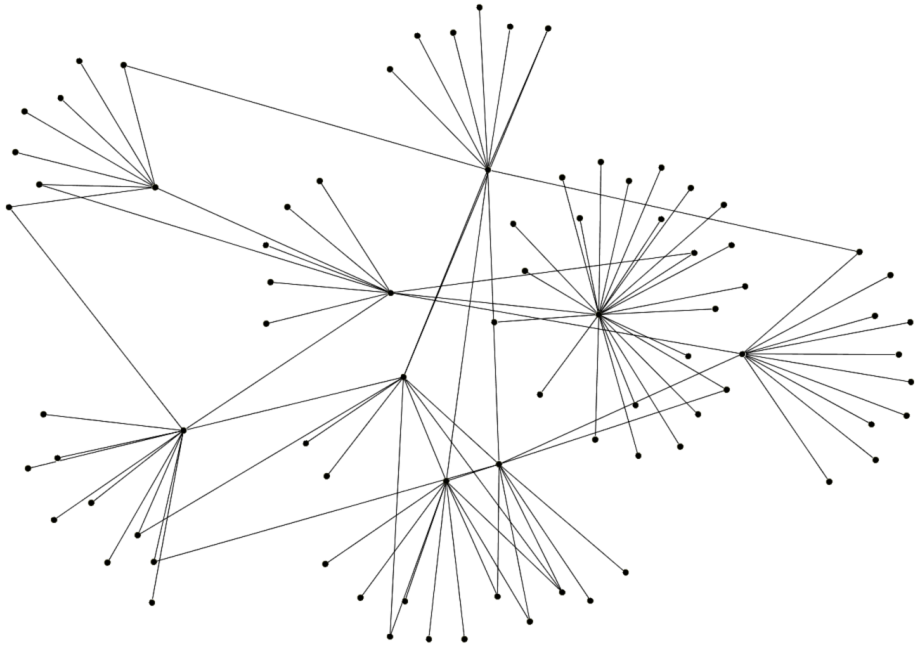


Fig. 10. Celeste network consisting of 82 nodes

But when a more complex network has to be drawn, comprehensibility suffers from too many edge crossings. Fig. 11 shows a network of 131 nodes. The center area of the graph gets cluttered with so many edges that it is difficult to recognize the network structure.

This problem can be partially solved by showing only those edges adjacent to a selected focal node, as seen in Fig. 12.

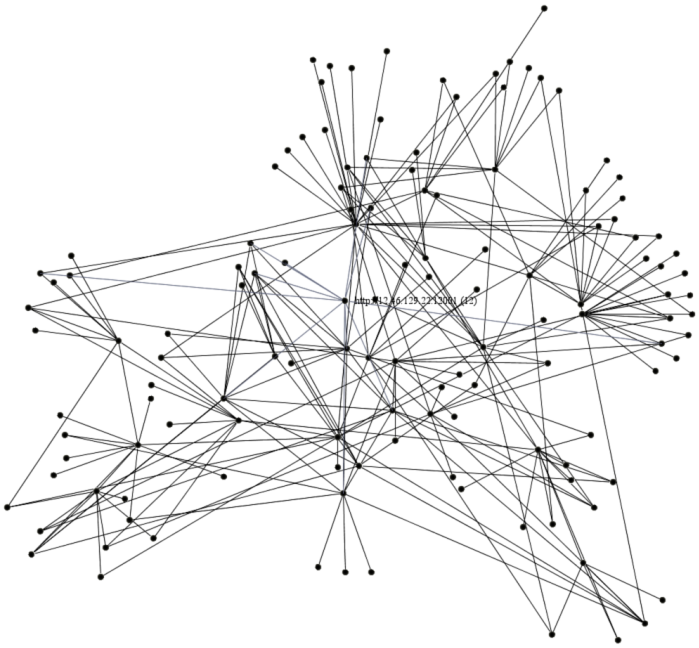


Fig. 11. Celeste network consisting of 131 nodes

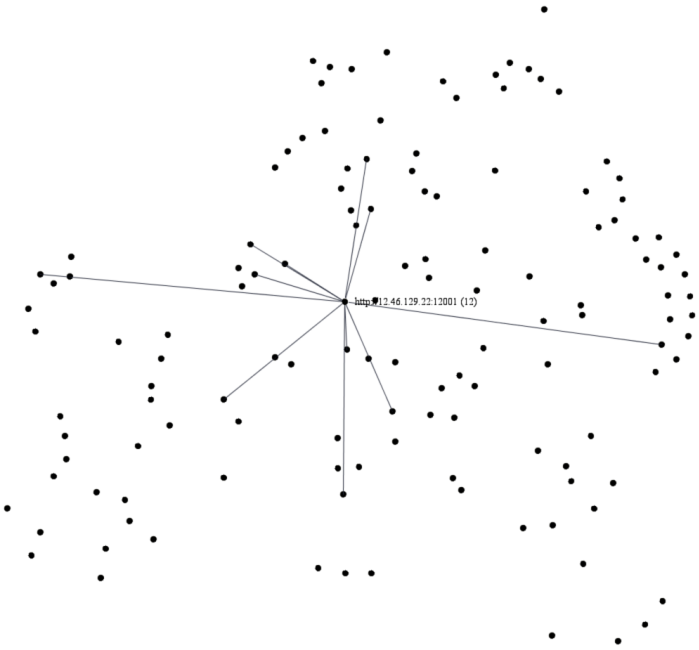


Fig. 12. Network consisting of 131 nodes with only the edges adjacent to a selected focal node visible

5 Conclusions and possible future work

In this paper, we examined several methods to display graph data in a comprehensible form. The different methods were evaluated for their suitability for the given data structures of *Celeste Visualizer*.

A force-based algorithm was chosen for implementation. Test runs on the Celeste network were conducted using PlanetLab [14]. The results were satisfactory, but room for further improvements is left.

An obvious task would be to address the problem of unnecessary edge crossings. Optimally, the existing algorithm would be combined with a planarization algorithm, without losing the advantages of the force-based variant.

The computing time for the force-based algorithm could probably be sped-up by implementing a grid-method as briefly described in [9]. In this variant the screen is divided into a grid of squares. At each iteration, each vertex is placed in its grid square and repulsive forces are computed only between it and the vertices in the same and nearby squares. Since repulsive forces decrease as the inverse square of the distance, the computing error of omitting more distant vertices becomes marginal.

The node information displayed in the popup box is fairly basic in its current state. Since the method allows for arbitrary data to be displayed, it could be expanded to include more data about the nodes and overall network traffic.

Also the edges themselves could be used to represent additional info about the *Celeste* network. Each edge carries information about its color and length. Different colors could be used to visualize traffic between the nodes. The edge length could be used in the force-based algorithm, since each edge can have a different desired length. This desired length could be used to represent a metric, like the round trip time (RTT) between two adjacent nodes.

6 References

[1] G. Caronni, R. Rom, G. Scott. *Celeste: An automatic storage system, white paper*

[2] J. Hopcroft and R.E. Tarjan. *Efficient planarity testing*. J. ACM, 21(4):549-598, 1974

[3] H. C. Purchase, R.F.Cohen, M.James. *Validating graph drawing aesthetics*. Proc. Graph Drawing '95, LNCS, 1027:435-446, 1996

[4] D. R. Wood. *Drawing planar graphs*. STAN-CS-82-943, Stanford University, 1981

[5] N. Chiba, K. Onoguchi, T. Nishizeki. *Drawing planar graphs nicely*. Acta Informatica, 22:187-201, 1985

[6] W. Schnyder. *Embedding planar graphs on the grid*. Proc. 1StACM-SIAM Symp. On Discrete Algorithms, 138-147, 1990

[7] K. Sugiyama, S. Tagawa, M.Toda. *Methods for visual understanding of hierarchical systems*. IEEE Trans. Syst. Man Cybern,, SMC-11(2):109-125, 1981

[8] F. J. Brandenburg, M. Juenger, P. Mutzel. *Algorithmen zum automatischen Zeichnen von Graphen*. Informatik Spektrum. 20(4):199-207, 1997

[9] T. M. J. Fruchterman and E.M. Reingold. *Graph Drawing by Force-directed Placement*. Software – Practice and Experience, Vol. 21(11), 1129-1164, Nov. 1991

[10] P. Mutzel. *Zeichnen von Diagrammen – Theorie und Praxis*. MPI für Informatik, Saarbrücken

[11] P. Eades. *A heuristic for graph drawing*. Congressus Nutnerantiunt, 42, 149-160, 1984

[12] W. Chen. *Graph theory and its engineering applications*, ISBN 981-02-1859-1, 1995

[13] A. Frick, A. Ludwig, H. Mehldau. *A fast adaptive layout algorithm for undirected graphs*, Fakultät für Informatik, Universität Karlsruhe

[14] PlanetLab Homepage: http://www.planet-lab.org/

Appendix A

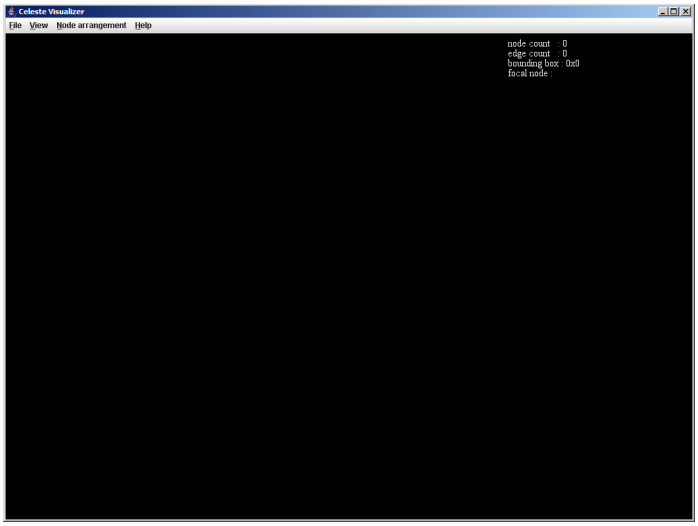
User guide

1 - Introduction

Welcome to Celeste Visualizer.

Celeste Visualizer is a tool which lets you connect to the Celeste network and displays all currently active nodes. Furthermore you can view additional data of any particular node.

When you start Celeste Visualizer, you see a black window (the canvas) and a menu bar on top.



2 - The "File" menu

This menu lets you connect/disconnect from the Celeste network.

To connect to the Celeste network, select **"Add node by URL"** and enter the URL of any Celeste node in the appearing popup dialog.

To stop Celeste Visualizer from scanning for more nodes, select **"Stop node explorer"**. Use **"Clear nodes"** to remove all nodes from the canvas.

"Quit Visualizer" ends the application.

3 - The "View" menu

This menu lets you change various settings of what is displayed.

"Show bargraph" toggles on/off a diagram on the right side of the canvas, showing the number of connections each node has.

"Show rings" displays little rings around each node, it has no practical use.

"Show all edges" causes all edges to be displayed. This can cause some slowdown, if there are too many edges (depending on your machine).

"Continuous" makes the "Brownian" and "Random" animations to run continuously.

"Info" toggles the name display for all nodes on/off.

"Undulate" lets all nodes undulate and change color.

"Brownian" lets all nodes randomly move around similar to Brownian motion.

"Fit to window" resizes the whole set of nodes so it fits into the canvas.

"Set animation speed" lets you change the animation speed. Values between 0 an 1 can be entered, with 0 stopping all animations and 1 causing instantaneous movement.

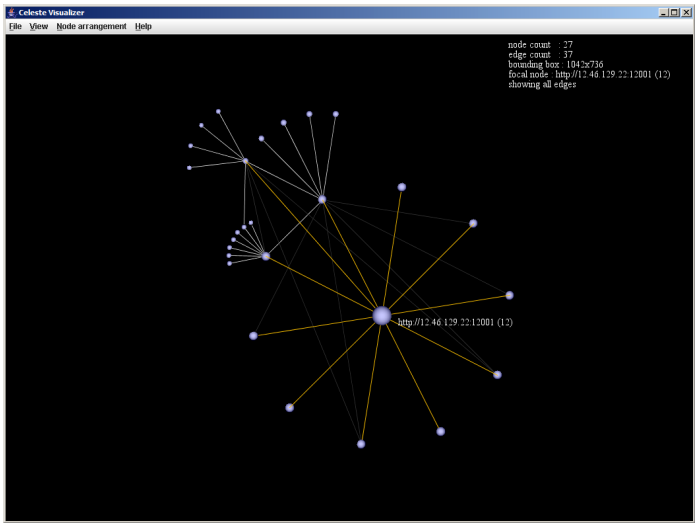
"Save a screenshot" saves a screenshot of the canvas. It is written in PNG format to the folder from which you ran Celeste Visualizer.

4 - The "Node arrangement" menu

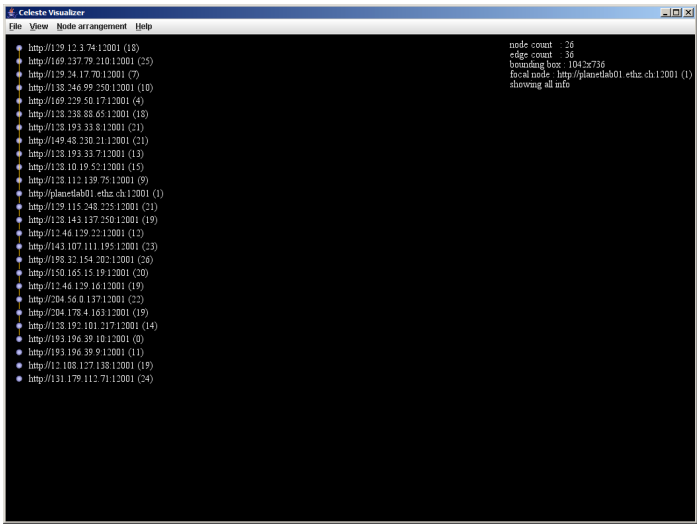
This menu lets you select in which way the nodes are being arranged in the canvas. Select one of the following options to change the node arrangement. The default option is **"Tiled"**.

"Random" simply distributes the nodes randomly in the canvas.

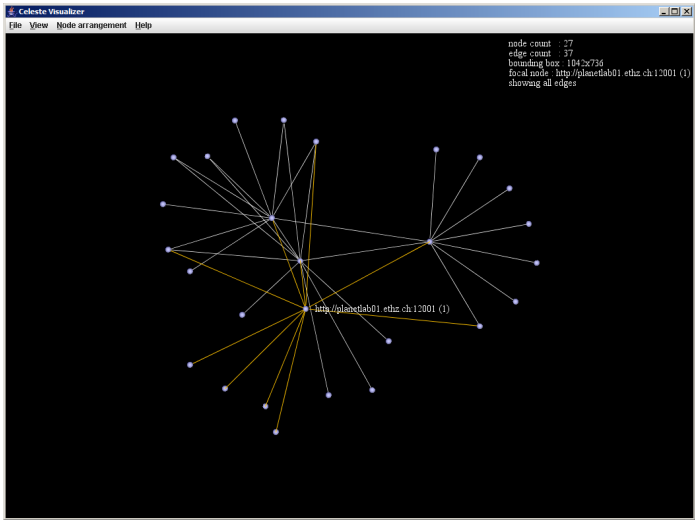
"Circular" distributes the nodes in a circular way growing outwards from a focal node. You first have to choose a focal node by left-clicking on any node.



"Tiled" lists the nodes starting from the upper left corner. If **"Info"** is toggled on, some space for the node names is left between the columns.



"Force Based" arranges the nodes according to a physical model. The nodes repel each other, while edges act as springs pulling adjacent nodes together. This option may take a few seconds to compute, if there are many nodes, depending on your machine.



5 - Navigating the canvas

When you successfully have connected to the Celeste network, nodes start appearing in the canvas. You can use the arrow keys to move the canvas around. Besides resizing the whole field to fit in the window, you can zoom in/out with the '.' and ',' keys respectively.

When right-clicking on a node, a popup appears showing additional info about that node. Right-click anywhere on the black background of the canvas to make the popup disappear.

Left-clicking on the somewhere background causes the bounding box for all nodes to be redefined to the rectangle from the upper left corner to the clicked point.