Visualizing Large and Clustered Networks

Katharina A. Lehmann and Stephan Kottler

University of Tübingen, Wilhelm-Schickard-Institute, Sand 14, 72076 Tübingen, Germany

Abstract. The need to visualize large and complex networks has strongly increased in the last decade. Although networks with more than 1000 vertices seem to be prohibitive for a comprehensive layout, real-world networks exhibit a very inhomogenous edge density that can be harnessed to derive an aesthetic and structured layout. Here, we will present a heuristic that finds a spanning tree with a very low average spanner property for the non-tree edges, the so-called *backbone* of a network. This backbone can then be used to apply a modified tree-layout algorithm to draw the whole graph in a way that highlights dense parts of the graph, so-called clusters, and their inter-connections.

1 Introduction

At first glance it seems prohibitive to visualize large and complex networks. The idea to represent these networks by suitable spanning trees and draw these trees instead of the whole graph, is a well-known approach, found, e.g., in [3,9,5]. In most of these cases it was assumed that the spanning tree was either given by the user or that the graph to draw was hierarchically organized and thus a spanning tree could be easily and more or less unambiguously derived. Here we will show that also the visualization of non-hierarchical networks is feasible with a spanning tree approach if the networks are clustered instead. A network is clustered if it can be decomposed into dense subgraphs that are only sparsely interconnected. In the past, this property has been used in various approaches, e.g., to analyse protein-protein-interaction networks or various social networks to find semantically connected subsets of vertices [4,8,13,12], to name but a few. We will show here, that this property can also be used to find a clear and computationally feasible layout for clustered graphs with more than 1,000 vertices and more than 10,000 edges. Actually computing a good partition can be computationally prohibitive, so our motivation is to decompose the graph into a set of *local* edges that are likely to be within clusters and a set of *global* edges that are likely to be between clusters. The decomposition into these sets has already been proven useful for drawing power-law graphs where the decomposition is derived by solving a network flow problem [1]. Our decomposition technique is based on finding a spanning tree that minimizes the distances between any two vertices connected by a non-tree edge, a so-called *backbone* of the graph. An edge whose endpoints have a large distance in the tree will be considered a global edge. We could show that finding the minimal spanning tree with this respect is

M. Kaufmann and D. Wagner (Eds.): GD 2006, LNCS 4372, pp. 240–251, 2007.

NP-hard by a reduction from *exact 3 cover*[1] [7]. Here, we will thus present two heuristics that yield very good initial backbones and an optional optimization step that can be used to improve the result.

A simple idea to draw a large graph is to take such a backbone and draw it with a tree layout algorithm, ignoring all non-tree edges. In most cases, this does not result in satisfying drawings. In our approach, non-tree edges will influence the order in which the children of a tree node are sorted, depending on the length of these edges in the backbone. This approach results in aesthetic drawings that reveal the large scale structure of the graph.

The paper is organized as follows: In Sec. 2 the needed definitions are given, the description of suitable backbones is given in Sec. 3. In Sec. 4 we then present a novel approach to draw large graphs based on backbones. We finish with a summary in Sec. 5.

2 Definitions

A graph is a pair (V, E) with V the set of vertices and $E \subseteq V \times V$ the set of edges, with $n := |V|$ the number of nodes, and $m := |E|$ the number of edges. We will assume that all graphs are free of self-loops, single-edged, undirected, and connected. The *neighborhood* $N(v)$ of a vertex v is given by $N(v) := \{w|(v, w) \in E\}$ and its *degree* $deg(v)$ by the cardinality $|N(v)|$ of its neighborhood. A *path* $P(s, t)$ between vertices s and t is a set of edges $\{e_1, e_2, \ldots, e_k\} \subseteq E$ such that $e_1 = (s, v_1)$, $e_k = (v_{k-1}, t)$, and for all $1 < i < k$: $e_i = (v_{i-1}, v_i) \in E$. The *path length* of a path $P(s, t)$ in an unweighted graph is given by the number of edges k in it. The *distance* $d(s, t)$ between two vertices s, t is given by the minimal length of any path between them if existent and ∞ otherwise. $d_{E'}(s, t)$ denotes the distance of two vertices using only the edges in $E' \subseteq E$. A graph is a *tree* if there is exactly one path between any pair of vertices. A *spanning tree* T of G is here defined as a subset of edges that constitutes a tree on V.

In the following, we will often use the term *cluster*. We use this term as an abbreviation for *dense subgraph*. In this work we will not give a proper distinction when a subgraph will be called a cluster and when not but rather speak of subgraphs that are *more clustered* than others.

3 The Backbone of Complex Networks

To harness the clustered structure of a large graph for computing a layout, we will use an approach that is based on finding a good spanning tree of the graph: Let T be a spanning tree of G that defines weights $\omega_T(e)$ for all edges $e = (v, w) \in E(G)$ in the following way:

$$\omega_T((v, w)) = d_T(v, w) \tag{1}$$

$d_T(e)$ will also be called the *tree distance* of edge e. The quality $Q(T)$ of a spanning tree will be measured by the sum of the weights it assigns to the edges:

[1] Unpublished result by KAL and M. Kaufmann.

$$Q(T) = \sum_{e \in E(G) \setminus T} \omega(e) \tag{2}$$

The motivation behind this quality measure is that, given a dense cluster of the graph, $Q(T)$ will in most cases be smallest, if all vertices of this cluster are in a small and contiguous subtree of T. Otherwise, all edges between these vertices would have high tree distance values. In other words, the lower $Q(T)$ is, the more non-tree edges are 'local' edges between vertices that are not far away in the tree. Thus, a spanning tree with a low $Q(T)$ can be called a *backbone* of the graph since it represents clusters in a concentrated way. Since trees are planar, the hope is that most 'local' edges will also span short distances if they are added to a drawing of the backbone.

A trivial lower bound for $Q(T)$ is given by $2(m - n + 1)$. This lower bound is for example met by a clique if the spanning tree consists of one vertex and all incident edges. The following procedure computes a non-trivial lower bound that depends on the structure of the given graph: For every edge $e = (v, w)$ the distance $d_{E \setminus \{e\}}(v, w)$ is computed. Let $\Sigma(G)$ denote the sum of the $m - (n - 1)$ lowest values of $d_{E \setminus \{e\}}(v, w)$.

Lemma 1. *$\Sigma(G)$ is a lower bound for $Q(T)$ for any spanning tree T in G.*

Proof. Let T^* denote an arbitrary spanning tree with minimal $Q(T^*)$. Let e be one of the $n - 1$ edges in T^*, then its weight does not contribute to $Q(T^*)$. If $e = (v, w)$ is not in T, $d_T((v, w))$ cannot be smaller than $d_{E \setminus \{e\}}(v, w)$. Since we do not know which edges will be in T^*, we disregard the $n - 1$ highest values of $d_{E \setminus \{e\}}(v, w)$ and thus, $\Sigma(G)$ is a lower bound for $Q(T^*)$. □

The quality of spanning trees with respect to $Q(T)$ can be very different, a fact that is shown in Fig. 1. Since finding the spanning tree with minimal $Q(T)$ is NP-hard as stated above we will now show greedy algorithms that compute reasonable initial backbones that can subsequently be improved by a local optimization heuristic.

3.1 Computing an Initial Backbone

To construct a backbone, the most simple idea is to choose one vertex at random and start a breadth first search and to mark the edge by which a vertex is first explored as tree edge. The quality $Q(T)$ of the resulting backbone is reasonably good compared to the above proposed quality measure Σ_G and the tree can be computed in $O(m)$. We will introduce two other methods that are computationally more involved but yield much better backbones in practice. Both heuristics grow a spanning tree S incrementally by first choosing the next vertex v to append to S and then choosing the best edge to hook v into S. Both start with one vertex chosen at random. With S the set of vertices already in the tree, let R denote the set of vertices $v \in G \setminus S$ directly connected to at least one node in S. The vertex to append next is the vertex with maximal degree of R, where ties are broken in favor of the vertex with maximal number of neighbors in S; remaining ties are then broken at random. The intuition behind this heuristic

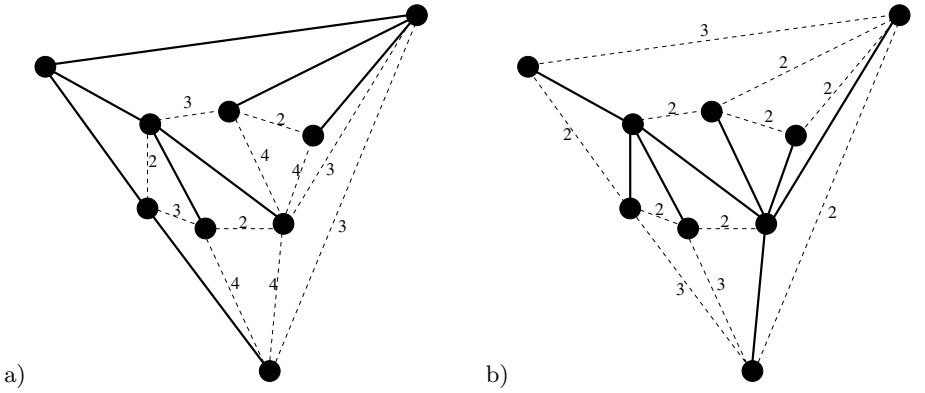

Fig. 1. The thick lines denote two different spanning trees T for the given graph. Numbers next to a (dotted) non-tree edge denote the tree distance of this edge. The spanning tree in a) has a quality $Q(T)$ of 34 and the spanning tree in b) has a $Q(T)$ of 25.

is that vertices that are appended early to the growing backbone will influence the backbone's structure most. Since a vertex with a high degree will contribute a large sum of backbone distances to $Q(T)$, these nodes should have a large influence and thus be appended as early as possible. A trivial implementation searches for the vertex to append in $O(n)$ in every step, yielding a runtime of $O(n^2)$ for all steps. A more sophisticated data structure that keeps vertices in R sorted in a kind of two-dimensional array of lists, can reduce this runtime to $O(n\ deg^*)$, where deg^* is the maximal degree in the graph. For very large real-world networks this is in most cases a significant improvement.

In general, the chosen vertex v will have more than one neighbor in S and its tree edge will connect it to one of them. These neighbors are the possible *hooks* of v. Note that by choosing one of the edges to some hook to be v's tree edge, the tree distances of all the possible tree edges of v are determined. Thus, the first variant, the *minimized inner distance tree*, will choose that hook that minimizes the tree distances of all the other possible tree edges:

Minimized Inner Distance Tree. Let $S(v)$ denote the neighbors of the chosen vertex v in S, i.e., the hooks of v. Since only one of the edges incident to a hook can be a tree edge without inducing a cycle in T, it is necessary to choose the one hook h^* that minimizes the tree distances of all the other edges to hooks. Thus, for every hook the distance to all other hooks is summed up and the edge to the hook with the minimal distance to all other hooks is chosen as new tree edge (Fig. 2 **a**).

By holding an array $D(T)$ of size n^2 that keeps the distance $d_T(s,t)$ for all vertices s,t in S, this computation can be done in $O((deg^*)^2)$. After the best hook h^* has been chosen, this data structure has to be updated by adding the distances $d_T(v,w)$ between the newly added vertex v and all other vertices w in S to $D(T)$. Since $d_T(v,w) = d_T(h^*,w) + 1$ for all $w \in S$, this can be done in

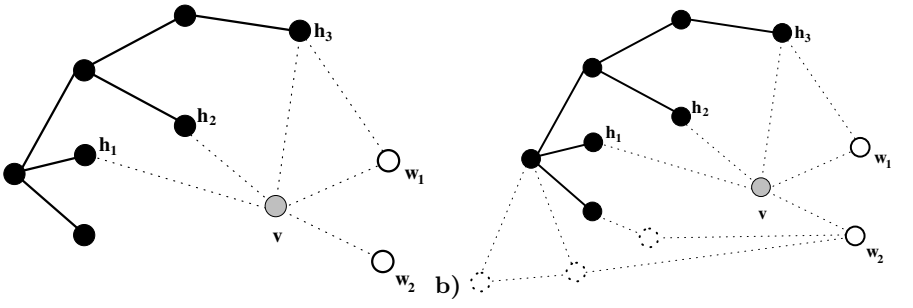

Fig. 2. a) Minimized Inner Distance Tree: Entering node v has three hooks h_1, h_2, h_3. h_2 minimizes the sum of the tree distances of v's edges to h_1, h_3 with a sum of 8, and thus h_2 is h^*. **b)** The tree distance of v's edges to w_1, w_2 can be estimated by determining the distance of the hooks to these neighbors. It follows that h_3 has the best sum of distance to all others: $|P(h_3, h_1)| = 4$, $P(h_3, h_2)| = 3$, $|P(h_3, w_1)| = 5$, $|P(h_3, w_2)| = 1$.

$O(n)$. Thus, the entire runtime to construct a minimized inner distance tree is given by $O(n(deg^*)^2 + n^2)$.

Lemma 2. *A minimized inner distance tree for some randomly chosen root node can be computed in* $O(n(deg^*)^2 + n^2)$.

While this tree only regards those (inner) edges to other vertices in S, the next one tries to estimate the tree distance of the other edges of v as well:

Minimized Entire Distance Tree. Let again $S(v)$ denote the neighbors of the chosen vertex v in S, and $N(v)$ denote the full neighborhood of v in G. For those edges of v that do not lead directly to vertices in S, it is hard to estimate their tree distance: It could be that they will later choose v as their hook to the growing tree and in this case an edge will not contribute to $Q(T)$. Since it is unlikely that all of them will use the edge to v as their tree edge, it would be good to choose a hook h^* such that all neighbors w of v have a short alternative path $P'(h^*, w)$ to v: A path $P'(h, w)$ is considered as an alternative if it can be split into two paths, the first using only vertices of S, the second -if necessary- only vertices of $V \backslash S$. In this way, the currently known structure of the tree is used as much as possible and the edges that are not yet known to be in the tree are only used for the last bit to reach w (Fig. 2). With this intuition, we will choose the hook $h^* \in S(v)$ that minimizes the following sum:

$$\sum_{w \in N(v)} |P'(h, w)| \tag{3}$$

Note that the sum in Equ. 3 contains also the sum of the inner distances and thus the name of the tree is justified. A summary of the attachment procedure *hookIntoTree*(E, T, E_S, v) is given in [10].

This computation can be done by computing the distance of all vertices to every hook of v which can be accomplished in $O(m\ deg^*)$. It follows that a minimized entire distance tree can be computed in $O(nm\ deg^*)$.

Lemma 3. *A minimized entire distance tree for some randomly chosen root can be computed in* $O(n\ deg^* m)$.

Table 1 shows a comparison of all three trees for some real-world networks. It is clearly visible that the higher computational effort for minimized inner distance and minimized entire distance trees results in much better backbones than the simple BFS tree and come near to the lower bound given by $\Sigma(G)$. However, even a good initial backbone can still be improved by the following optimization heuristic.

3.2 Optimization of the Backbone

The following steps allow for a local optimization of the initially computed backbone T. The main idea is that any edge e that is not in T would induce a cycle if it was added to T. By removing any other edge f of this cycle, a new spanning tree $T'(e, f) := (T \cup e) \backslash f$ results. If no ambiguity is given we will reduce $T'(e, f)$ to T' in the following. If $Q(T')$ is smaller than $Q(T)$, than e should replace f in T. We will call e the *entering edge* and f the *leaving edge*. To analyze whether $Q(T')$ is smaller than $Q(T)$, the following definitions are helpful: Let e be any non-tree edge, then $P_T(e)$ denotes the path in T that connects the end vertices of e, the so-called *tree path* of e.

Proposition 4. *For all non-tree edges* i *with* $f \notin P_T(i)$, $d_T(i)$ *will not be changed.*

Proof. Since all edges of $P_T(i)$ are still in T, $d_T(i)$ cannot be increased. Let's assume that $d_T(i)$ is decreased by the insertion of e. This means that there is a second path connecting the end vertices of i, violating the tree property of T. □

Let i denote some non-tree edge whose tree path contains at least one of the edges of $P_T(e)$, and let $C_T(i, e)$ denote the set of shared edges:

$$C_T(i, e) := P_T(i) \cap P_T(e) \tag{4}$$

If the leaving edge f is in this set, the tree path of i will be altered. To describe the change, the following definitions are needed (Fig. 3 **a**): Let $C_T(e)$ denote all edges in the cycle that is introduced by adding e to T. Note that $C_T(e)$ is given by $P_T(e) \cup \{e\}$. Let $\overline{C_T(i, e)}$ denote the complement of $C_T(i, e)$ in cycle $C_T(e)$. The new tree path $P_{T'}(i)$ is then given by

$$P_{T'}(i) = P_T(i) \cup \overline{C_T(i, e)} \setminus C_T(i, e). \tag{5}$$

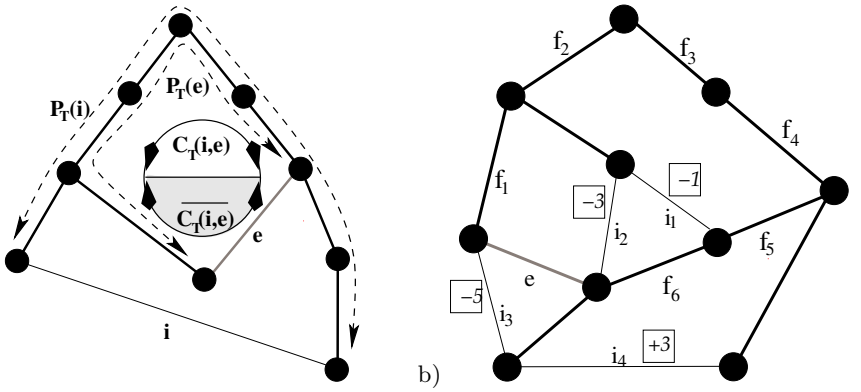

Fig. 3. a) e is the entering edge, the tree paths $P_T(e)$ and $P_T(i)$ of some other non-tree edge i are indicated by the dotted arrows. Every non-tree edge i with $C_T(i,e) \neq \emptyset$ will have to change its tree path if the leaving edge is element of $C_T(i,e)$. The new tree path is built by removing from the old tree path all edges from $C_T(i,e)$ and adding the complement of the circle, i.e., $\overline{C_T(i,e)}$, to it.**b)** Again, e is the entering edge, i_j are edges that could be affected by choosing some of the possible leaving edges f_i. The boxed numbers give the difference between the new and old tree distance. It follows that for entering edge e, f_2, f_3, or f_4 would yield the best optimization with a value of $\Delta Q(T,e,f)$ of -9.

Note that this new tree path is always the same for any fixed non-tree edge i, independent of the identity of the leaving edge f as long as $f \in C_T(i,e)$ (s. Fig. 3 **b**). Thus, $\Delta d_T(i,e) := d_{T'}(i) - d_T(i)$ is given by:

$$\Delta d_T(i,e) = |\overline{C_T(i,e)}| - |C_T(i,e)| \tag{6}$$
$$= |C_T(e)| - 2|C_T(i,e)| \tag{7}$$

With $I_e(f)$ denoting the set of non-tree edges i with $f \in C_T(i,e)$, we can now state the following lemma:

Lemma 5. *For fixed entering edge e and leaving edge f, the difference in $Q(T)$ denoted by $\Delta Q(T,e,f)$ can be computed by:*

$$\Delta Q(T,e,f) = \sum_{i \in I_e(f)} \Delta d_T(i,e,f) \tag{8}$$

$\Delta(Q(T,e,f))$ can be computed efficiently by first determining the set $I(e) = \cup_{f \in P_T(e)} I_e(f)$ of all edges i that are depending on at least one edge of $C_T(e)$ in their tree path. This can be done very efficiently if every tree edge f stores $I_e(f)$ in a bit map. A bit map allows space and time efficient set operations, e.g., conjunctions and disjunctions. With at most n sets $I_e(f)$, the set $I(e)$ can be computed in $O(nm)$. The tree path $P_T(i)$ of every non-tree edge i is also

stored as bits in a bit map. By simple $OR-$, $XOR-$, and AND-Operations all required sets $C_T(i,e)$, $\overline{C_T(i,e)}$, and $\Delta d_T(i,e)$ can be computed in $O(m)$ for a single non-tree edge i and in $O(m^2)$ for all of them. The leaving edge is the edge f with minimal $\Delta Q(T,e,f)$, which can be computed in $O(nm)$ where ties are broken at random. If there is no leaving edge because all resulting trees T' would be worse, nothing will happen and the next entering edge e is chosen at random. A summary of this algorithm is given in [10]. After e and f have been chosen in this way, some updates have to be done that are also computed very efficiently by operations on the bit maps. These updates can then be computed in $O(m^2)$.

Lemma 6. *A single local optimization step can be computed in* $O(m^2)$.

Table 1 shows that the optimization is able to decrease the already good $Q(T)$ of an *minimized entire distance tree* significantly towards the lower bound. The table also gives the time spent on the optimization, showing that there is a trade-off between the wanted quality of the backbone and the time spent on its computation.

4 Using the Backbone for Computing a Layout

As indicated above, a good backbone will try to concentrate the vertices of any cluster on a small, connected subtree. By doing so, the tree also indicates that edges with a high tree distance are more likely to be inter-cluster edges. These properties of the backbone can be used for computing a layout that co-locates the vertices that are supposedly in a dense part of the graph and simultaneously highlights the inter-connections between these dense parts.

To harness the backbone, our layout approach is based on a tree layout that is adapted towards the needs of a full graph. The layout of the graph can be computed by a variation of the balloon tree layout [3], resulting in a drawing which we will call a *backbone balloon drawing*. In the original balloon drawing of a tree, every subtree is enclosed entirely in a circle that is positioned in a wedge whose end-point is the parent node of this subtree. The radius of each circle is proportional to the number of vertices in the subtree.

To adapt this tree layout towards the needs of a full graph, the basic idea is to use the backbone and compute a balloon drawing for it and re-insert all non-tree edges as straight lines. To make this drawing a good drawing for the whole graph, the only parameter to change is the order of the children of any vertex in the tree. Since all direct neighbors of any vertex in the tree are positioned in a circle, the order of these children can be determined by a variation of the algorithm for crossing reduction in circular layouts [2]. The original algorithm is composed of two phases: In the first phase an initial ordering is heuristically determined. This is optimized by subsequent rounds of local sifting, where each vertex can try to improve the number of crossings by changing its position in the order computed so far. The application of this algorithm in a backbone balloon drawing requires the following two modifications:

1. Every edge between the children of a vertex in the tree can not only cross with each other, but also with the spokes, i.e., the edges from the father to its children. This changes the computation of the resulting number of crossings slightly.
2. Let $T(v)$ and $T(w)$ denote the subtrees rooted at v and w, respectively, and let v and w be children of the same vertex. If the number of edges between these subtrees is large, then v and w should be close in the resulting order which is of course not regarded in the original algorithm.

The second point can be dealt with by introducing additional edges between any two children v, w whose subtrees are connected by edges. Additionally, all edges will be assigned weights that present the number of edges between $T(v)$ and $T(w)$. The weight of a crossing between two edges is now given by the sum of the weights of the crossing edges, and the optimization goal is to minimize the sum of the weights of all crossings and not to minimize the number of crossings. The weights of all the edges between any two children can be computed in $O(nm)$. Every round of local sifting in a given circle with at most deg^* vertices can be computed in $O((deg^*)^2)$ as shown in [2]. Since there are at most n circles in the drawing, this sums up to $O(n(deg^*)^2)$ which is the largest factor in computing the backbone balloon drawing.

4.1 Experiments

We have applied the above presented variant of the balloon layout algorithm on different types of networks, shown in detail in [10]. Here, we show exemplary one network, a so-called *Amazon recommendation* network. To derive it, we start at some book that is offered by the Internet bookshop www.amazon.com and follow the links presented under the title "customers who bought this book also bought". By recursively following these links, very large and complex networks can be created. By construction, the outdegree of every vertex in the network is bounded by 6. The network shown here starts at [11] (Fig. 5). The balloon tree drawing shows discernible clusters connected by long-range edges, that are even more pronounced in the drawing that is based on an optimized backbone with minimized entire distance. This visual impression is supported by the fact that the force-directed drawing has the highest (normalized) total edge length of 434813, the one based on the unoptimized backbone has a total edge length of 321292 and the one based on the optimized backbone has a total edge length of 220857.

To show the quality of the different backbone heuristics and the optional optimization step, we have conducted experiments on this Amazon recommendation network and two other networks, shown in Table 1. For the creation of the Live Journal network a crawl was started at some participant of www.livejournal.com, following the links to designated friends unto depth 3. The co-authorship network is described in [14]. Fig. 4 gives a showcase for the improvements of $Q(T)$ by the optimization heuristic. It is clearly visible that the time spent in this step is worth the effort.

Table 1. For every network, 10 instances of every kind of spanning tree were computed. Displayed is the average $Q(T)$, its deviation, and the average time and its deviation to compute the tree. Note that the best unoptimized spanning trees already have a quality that is close to the lower bound given by $\Sigma(G)$ that can nonetheless be further reduced by the optimization. Furthermore, every of those 10 instances started at another, randomly chosen start vertex. The low deviation in $Q(T)$ shows that the method gives a stable $Q(T)$, independent of the choice of the start vertex. The experiments were conducted on a Pentium 4 with 3.2 GHz and 2GB RAM.

Graph	**BFS**	**Minimized Inner Distance**	**Minimized Entire Distance**	**Optimized**	$\Sigma(G)$
Amazon recommendation network $n = 3437$ $m = 9671$	31342 ± 316 73 ± 11 [ms]	21819 ± 57 358 ± 34 [ms]	20654 ± 24 70 ± 0.3[s]	17596 ± 28 20min14s	12468
Live Journal $n = 3763$ $m = 11149$	29615 ± 1332 65 ± 11[ms]	23156 ± 71 284 ± 19[ms]	22058 ± 38 100 ± 0.4[s]	19588 ± 3 22min12s	14774
Co-Authorship Network $n = 12357$ $m = 19448$	52896 ± 1447 131 ± 18 [ms]	52463 ± 222 480 ± 34[ms]	49951 ± 98 337 ± 0.6[s]	34287 ± 47 49min2s	14184

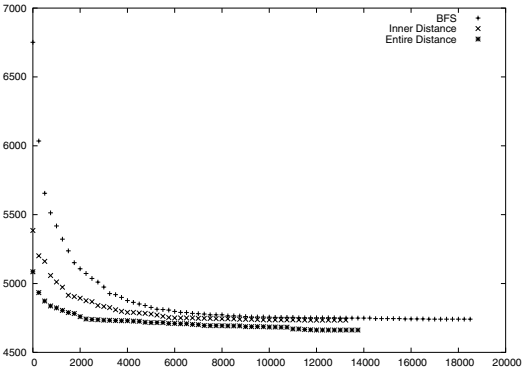

Fig. 4. For a smaller Amazon recommendation network with $n = 852$ and $m = 4220$, one BFS, one minimized inner distance and one minimized entire distance tree was computed and improved by the optimization heuristic until no further improvements could be found, i.e., a local minimum is reached. The trees start and end with the following $Q(T)$'s: 6572/4641 (BFS), 5385/4734 (Inner), and 5085/4662 (Entire), respectively. Note that $\Sigma(G)$ is 3822.

5 Summary

In this paper we presented a new quality measure $Q(T)$ for a spanning tree that helps to visualize large and clustered networks. We have shown that spanning

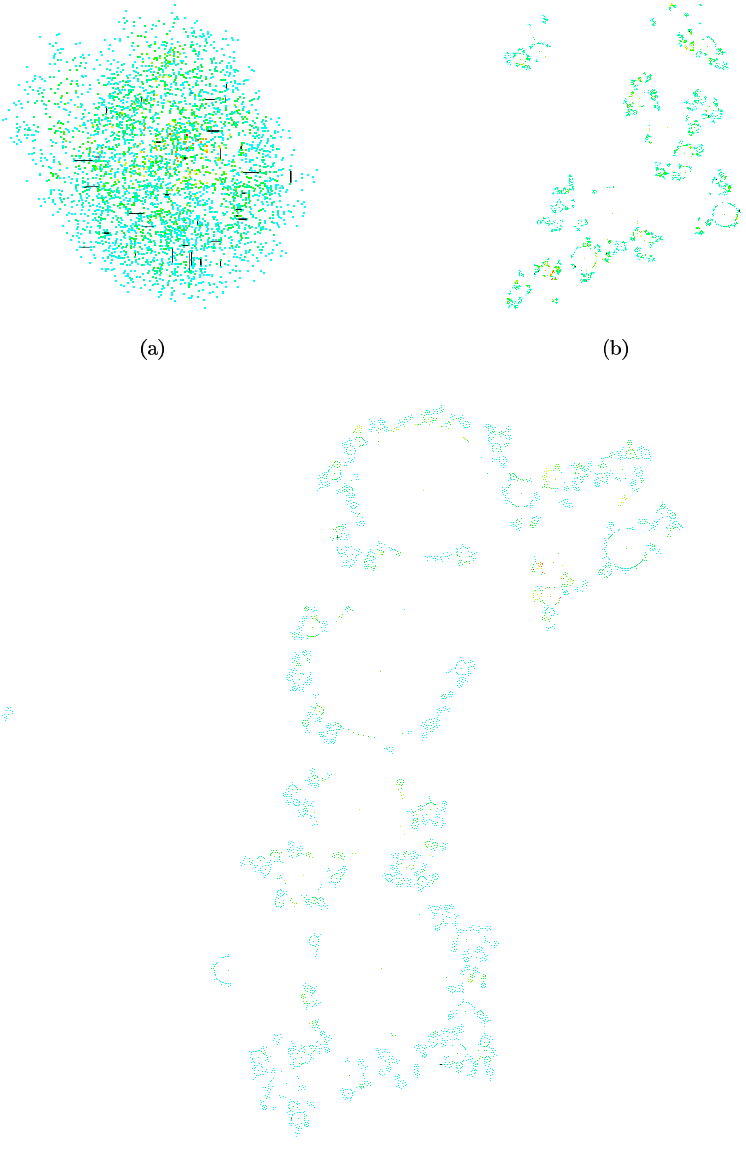

Fig. 5. a) A layout based on a force-directed approach, implemented by [15]. The normalized total edge length of this drawing is 434813. **b)** A balloon layout drawing based on a simple, unoptimized backbone with minimized entire distance. The normalized total edge length of this drawing is 321292. **c)** A balloon layout drawing based on an optimized backbone with minimized entire distance. The normalized total edge length of this drawing is 220857. The time to compute this layout was on average around 20 minutes (averaged over 5 drawings).

trees with a low $Q(T)$ can be computed in reasonable time and that these can be improved further by a local optimization heuristic. These trees or backbones can then be used to derive variations of classic layouts that are suitable for clustered graphs. Looking at the resulting drawings, a further application is to use these drawings as a basis for a geometric clustering method. First experimental evidence shows that a new, geometric variation of the Girvan-Newman-Clustering [8] applied to the drawings yields partition with a high modularity value [10] while being much faster computed than in the original approach. Further work will have to show whether backbones can also be used to adapt other drawings, such as the hierarchical Sugiyama drawing, or maybe build the basis for new approximation algorithms.

References

1. Reid Andersen, Fan Chung, and Linyuan Lu. Drawing power law graphs. In *Proceedings of the 12th Symposium on Graph Drawing (GD'04)*, 2004.
2. Michael Baur and Ulrik Brandes. Crossing reduction in circular layouts. In *Proceedings of the 30th Workshop on Graph-Theoretic Concepts in Computer Science (WG'04)*, 2004.
3. J. Carriére and R. Kazman. Interacting with huge hierarchies: Beyond cone trees. In *Proceedings of the ACM conference on Information Visualization 1995*, pages 74–81, 1995.
4. I. Derényi, G. Palla, and T. Vicsek. Clique percolation in random networks. *Phys. Rev. Lett.*, 94:160202, 2005.
5. Jean-Daniel Fekete, David Wang, Niem Dang, Aleks Aris, and Catherine Plaisant. Overlaying graph links on treemaps. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis'03)*, 2003.
6. T.M.J. Fruchtermann and E.M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.
7. Michael R. Garey and David S. Johnson. *Computers and intractability.* W.H. Freeman and Company, New York, 1979.
8. Michelle Girvan and M.E.J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99:7821–7826, 2002.
9. Ivan Herman, Guy Melançon, Maurice M. de Ruiter, and Maylis Delest. *Lecture Notes in Computer Science*, chapter Latour - A tree visualization system, page 392ff. Springer Verlag, Berlin, 2000.
10. Katharina A. Lehmann and Stephan Kottler *Visualizing Large and Clustered Networks.* Technical Report of the Wilhelm-Schickard-Institut, WSI-2006-06, ISSN 0946-3852, September 2006.
11. Mark Newman, Albert-Laszlo Barabasi, and Duncan J. Watts. *The structure and dynamics of networks.* Princeton University Press, 2006.
12. Andreas Noack. An energy model for visual graph clustering. In *Proceedings of the 11th International Symposium on Graph Drawing (GD'03)*, 2004.
13. Andreas Noack. Energy-based clustering of graphs with nonuniform degrees. In *Proceedings of the 13th International Symposium on Graph Drawing (GD'05)*, 2005.
14. G. Palla, I. Derényi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435:814, 2005.
15. OrganicLayouter in the yFiles library *Sebastian Mueller.* www.yworks.com, Version 2.2.