Drawing Large Graphs by Multilevel Maxent-Stress Optimization

Henning Meyerhenke[1], Martin Nöllenburg[2], and Christian Schulz[1(✉)]

[1] Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT),
Karlsruhe, Germany
{meyerhenke,christian.schulz}@kit.edu
[2] Algorithms and Complexity Group, TU Wien, Vienna, Austria
noellenburg@ac.tuwien.ac.at

Abstract. Drawing large graphs appropriately is an important step for the visual analysis of data from real-world networks. Here we present a novel multilevel algorithm to compute a graph layout with respect to a recently proposed metric that combines layout stress and entropy. As opposed to previous work, we do not solve the linear systems of the maxent-stress metric with a typical numerical solver. Instead we use a simple local iterative scheme within a multilevel approach. To accelerate local optimization, we approximate long-range forces and use shared-memory parallelism. Our experiments validate the high potential of our approach, which is particularly appealing for dynamic graphs. In comparison to the previously best maxent-stress optimizer, which is sequential, our parallel implementation is on average 30 times faster already for static graphs (and still faster if executed on one thread) while producing a comparable solution quality.

1 Introduction

Drawing large networks (or graphs, we use both terms interchangeably) with hundreds of thousands of nodes and edges has a variety of relevant applications. One of them can be interactive visualization, which helps humans working on graph data to gain insights about the properties of the data. If a very large high-end display is not available for such purpose, a hierarchical approach allows the user to select an appropriate zoom level [1]. Moreover, drawings of large graphs can also be used as a preprocessing step in high-performance applications [22].

One very promising class of layout algorithms in this context is based on the *stress* of a graph. Such algorithms can for instance be used for drawing graphs with fixed distances between vertex pairs, provided *a priori* in a distance matrix [13]. More recently, Gansner et al. [12] proposed a similar model that includes besides the stress an additional entropy term (hence its name *maxent-stress*). While still using shortest path distances, this model often results in more satisfactory layouts for large networks. The optimization problem can be cast

---

This is a short version of the technical report (TR) [27].
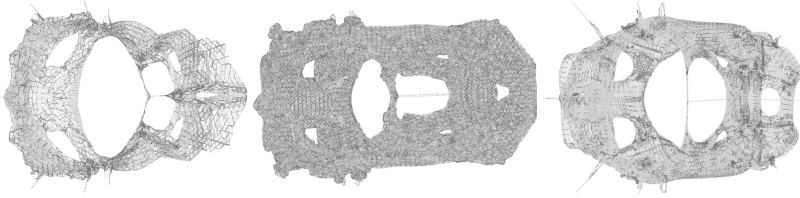
Fig. 1. Drawings of bcsstk31. Left to right: PivotMDS [5], Maxent [12], MulMent (new).

as solving Laplacian linear systems successively. Since each right-hand side in this succession depends on the previous solution, many linear systems need to be solved until convergence – more details can be found in Sect. 2.3.

Motivation. We want to employ this maxent-stress model for drawing large networks quickly. Yet, solving many large Laplacian linear systems can be quite costly. A conjugate gradient solver (used in [12]) is easy to implement but has superlinear running time. Solvers with provably nearly-linear running time exist but are not yet competitive with established methods in practice (see [18] for an experimental comparison). Multigrid methods [24,26] for Laplacian systems may seem appealing in this context, but their setup phase building the multigrid hierarchy can be expensive for large graphs.

Gansner et al. [12] also suggested (but did not use) a simpler iterative refinement procedure for solving their optimization problem. This procedure would be slow to converge if used unmodified. However, if designed and implemented appropriately, it has the potential for fast convergence even on large graphs. Moreover, as already observed in [12], it has high potential for parallelism and should work well on dynamic graphs by profiting from previous solutions.

Outline and Contribution. The main contribution of this paper is to make the alternative iterative local optimizer suggested by Gansner et al. [12] (for details on this and other related work see Sect. 2) usable and fast in practice. To this end, we design and implement a multilevel algorithm tailored to large networks with unit target edge lengths (see Sect. 3). The employed coarsening algorithm for building the multilevel hierarchy can control the trade-off between the number of hierarchy levels and convergence speed of the local optimizer. One property of the local optimizer we exploit is its high degree of parallelism. Further acceleration is obtained by approximating long-range forces. To this end, we use coarser representatives stored in the multilevel hierarchy.

Our experimental results in Sect. 4 show that force approximation rarely affects the layout *quality* significantly – in terms of maxent-stress values as well as visual quality, also see Fig. 1 and TR [27]. The parallel implementation of our multilevel algorithm MulMent with force approximation is, however, on average 30 times faster than the reference implementation [12] – and even our sequential approximate algorithm is faster than the reference. A contribution besides higher speed is that, in contrast to [12], our approach does not require input coordinates to optimize the maxent-stress measure.

2   Preliminaries

2.1   Basic Concepts

Consider an undirected, connected graph $G = (V, E, c, \omega, d)$ with node weights $c : V \to \mathbb{R}_{\geq 0}$, edge weights $\omega : E \to \mathbb{R}_{\geq 0}$, target edge lengths $d : E \to \mathbb{R}_{>0}$, $n = |V|$, and $m = |E|$. Often the function d models the required distance between two adjacent vertices. By default, our initial inputs will have unit edge length $d \equiv 1$ as well as unit node weight and edge weight $c \equiv 1$, $\omega \equiv 1$. However, we will encounter weighted problems in the course of our multilevel algorithm. Let $N(v) := \{u : \{v, u\} \in E\}$ denote the set of neighbors of v. A clustering of a graph is a set of *blocks* (= clusters) of nodes $\{V_1, \ldots, V_k\}$ that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. A *layout* of a graph is represented as a coordinate vector x, where x_v is the two-dimensional coordinate of vertex v. Since edges are drawn as straight-line segments between their incident nodes, x is sufficient to define the complete graph layout.

2.2   Related Work

Most general-purpose layout algorithms for arbitrary undirected graphs are based on physical analogies and can be grouped, according to Hu and Shi [19], into two main classes: algorithms in the *spring-electrical model* and algorithms in the *stress model*. Both classes of algorithms often yield aesthetically pleasing graph layouts that emphasize symmetries and avoid edge crossings at least in sparse graphs. Recent surveys of algorithms in these models are given by Hu and Shi [19] and by Kobourov [23].

In the spring-electrical model, first presented by Eades in 1984 [8], the analogy is to represent nodes as electrically charged particles that repel each other while edges are represented as springs exerting attraction forces to adjacent nodes. A graph layout is then seen as a physical system of forces and the goal is to find an optimal layout corresponding to a minimum energy state. Spring-electrical algorithms are also known as *spring embedders*, with the algorithm by Fruchterman and Reingold [10] being one of the most widely used spring embedder algorithms. It simulates the physical system of attractive and repulsive forces and iteratively moves each node into the direction of the resulting force. Each iteration requires, however, a quadratic number of force computations due to the repulsive forces between all pairs of nodes, which limits the scalability of the original approach. A faster approximative force calculation method based on quadtrees, aggregating especially the long-range forces, has been proposed by Barnes and Hut [3] and yields running times of $O(n \log n)$ under certain assumptions.

The (full) stress model is closely related to multidimensional scaling [25], and was introduced in graph drawing by Kamada and Kawai [21]. It is based on defining ideal distances d_{uv} not only between adjacent vertices but between all vertex pairs $(u, v) \in V \times V$ and then minimizing the layout stress $\sum_{u \neq v} w_{uv}(\|x_u - x_v\| - d_{uv})^2$, where w_{uv} is a weight factor typically chosen as $w_{uv} = 1/d_{uv}^2$. Often, the distance d_{uv} between adjacent nodes is set to 1, while the distance of non-adjacent nodes is the shortest-path distance in the graph.

Solving this model is typically done by iteratively solving a series of linear systems [13]. The need to compute all-pairs shortest paths and to store a quadratic number of distances again defeats the scalability of this original approach for large graphs. One of the fastest algorithms for approximatively solving the stress model instead is PivotMDS [5], which requires distance calculations from each vertex only to a small set of $k \ll n$ suitably chosen pivot vertices.

The stress model prescribes target distances not only for edges but for all vertex pairs. While this is a reasonable approach, it still brings artificial information into the layout process. An interesting alternative has been proposed by Gansner et al. [12]. Their algorithm (called *Maxent*) uses the sparse stress model, which only contains the stress terms for the edges of the graph. In order to deal with the remaining degrees of freedom in the layout, they suggest using the maximum entropy principle instead. Since our algorithm is closely related to Maxent, we discuss the latter in more detail in Sect. 2.3.

A general approach for speeding up layout computations for large graphs is the *multilevel technique*, which has been used in the spring-electrical [16,29,32] and in the stress model [11]. A multilevel algorithm computes a sequence of increasingly coarse but structurally related graphs as abstractions of the original graph. Starting from a layout of the coarsest graph, incremental refinement steps using the previous layout as a scaffold eventually produce a layout of the entire input graph, where the refinement steps are fast due to the good initial layouts. Hachul and Jünger [15] performed an extensive experimental evaluation of state-of-the-art layout algorithms for large graphs, including multilevel algorithms, and Bartel et al. [4] experimentally compared different combinations of coarsening, placement, and layout methods for the generic multilevel approach.

In addition to sequential algorithms for drawing large graphs, there is previous research in parallel layout algorithms, particularly using a graphics processing unit (GPU). Frishman and Tal [9] presented a multilevel force-based layout algorithm and implemented it using GPU-based parallelization. Ingram et al. [20] also exploit parallel GPU computations and presented a multilevel stress-based layout algorithm. Godiyal et al. [14] implemented a fast multipole algorithm on the GPU.

2.3   Maxent-Stress Optimization

Gansner et al. [12] proposed the maxent-stress model that combines a sparse stress model with an entropy term to resolve the degrees of freedom for non-adjacent vertex pairs. The entropy term itself is optimized when all nodes are spread out uniformly, similar to the repulsive forces in the spring-electrical model. Gansner et al. [12] showed that the maxent-stress model performs well on several measures of layout quality in distance-based embeddings and avoids typical shortcomings of other stress models, particularly for non-rigid graphs. Formally, the maxent-stress $M(x)$ of a layout x is defined[1] as

---

[1] In fact, Gansner et al. define a slightly more general model that considers the stress term for arbitrary supersets $S \supseteq E$ and allows variations of the entropy term. Our algorithm also works for the general model; to simplify the description, we restrict ourselves to the default model.

$$M(x) = \sum_{\{u,v\} \in E} w_{uv}(||x_u - x_v|| - d_{uv})^2 - \alpha \sum_{\{u,v\} \notin E} \ln ||x_u - x_v||, \qquad (1)$$

where d_{uv} is the target distance between nodes u and v and w_{uv} is a weight factor typically chosen as $w_{uv} = 1/d_{uv}^2$. Throughout the paper, we use this as a weight factor. The scaling factor α is used to modulate the strength of the entropy term and is gradually reduced in the implementation.

Gansner et al. minimize the maxent-stress using a technique that repeatedly solves Laplacian linear systems that additionally include a repulsive force vector which is approximated following the quadtree method of Barnes and Hut [3].

Alternatively, they proposed (but did not implement) the following local iterative force-based scheme to solve the maxent-stress model:

$$x_u \leftarrow \frac{1}{\rho_u} \sum_{\{u,v\} \in E} w_{uv} \left(x_v + d_{uv} \frac{x_u - x_v}{||x_u - x_v||} \right) + \frac{\alpha}{\rho_u} \sum_{\{u,v\} \notin E} \frac{x_u - x_v}{||x_u - x_v||^2}, \qquad (2)$$

where $\rho_u = \sum_{\{u,v\} \in E} w_{uv}$. Note that sometimes we use the abbreviation $r(u,v) := \frac{x_u - x_v}{||x_u - x_v||^2}$ and shortly call these values *r-values*.

3   Multilevel Maxent-Stress Optimization

As mentioned, a successful (meta)heuristic for graph drawing (and other optimization problems on large graphs) is the multilevel approach. We also employ this approach for maxent-stress optimization for several other reasons: (i) Some graphs (such as road networks) feature a hierarchical structure, which can be exploited to some extent by a multilevel approach and (ii) the computed hierarchy may be useful later on for multiscale visualization.

Before going into the details, we briefly sketch our algorithmic approach: The method for creating the graph hierarchy is based on fast graph clustering with controllable cluster sizes. Each cluster computed on one hierarchy level is contracted into a new supervertex for the next level. After computing an initial layout on the coarsest hierarchy level, we improve the drawing on each finer level by iterating Eq. (2). Additionally, this process exploits the hierarchy and draws vertices that are densely connected with each other (i. e. which are in the same cluster) close to each other.

3.1   Coarsening and Initial Layout

To compute the clustering we adapt size-constrained label propagation (SCLaP) [28], an algorithm originally developed for coarsening and local improvement during multilevel graph partitioning. SCLaP itself is based on the graph clustering algorithm *label propagation* [30]. The latter starts with a singleton clustering (i. e. each node is a cluster). The algorithm then works in rounds. Roughly speaking, in each round the algorithm visits all nodes in random order and assigns each node to the predominant cluster in its neighborhood. This way,

cluster IDs (= labels) propagate through the graph and nodes in a dense cluster usually agree on a common label.

However, clusters with unconstrained sizes are not desirable here since they would hamper convergence of the local improvement phase. The trade-off between this convergence speed and the number of hierarchy levels needs to be chosen properly for a fast overall running time. That is why SCLaP constrains cluster sizes, i.e. it introduces an upper bound $U := \max(\max_v c(v), W)$ on the cluster sizes (W is specified below), where constraining on the maximum node weight favors uniform coarsening. Consequently, in each SCLaP round, nodes are assigned to the predominant cluster that is not overloaded after the label change.

In our implementation, based on preliminary experiments, we set the parameter W to $\min(b^h, \frac{|V|}{f})$, where b and f are tuning parameters and h is the level in the hierarchy that we are currently working on. The intuition behind this choice is that we want the contraction process not to be too strong on the fine levels in order to allow fast convergence of local improvement algorithms, whereas we allow stronger contractions on coarser levels. If the contracted graph is not more than $10\,\%$ smaller than the graph on the current level, we decrease the value of f and set it to $0.7f$.

While the original label propagation algorithm repeats the process until convergence, SCLaP performs at most ℓ rounds, where ℓ is a tuning parameter. One round of the algorithm can be implemented to run in $\mathcal{O}(n + m)$ time.

Contracting a clustering works as follows: each block of the clustering is contracted into a single node. The weight of the node is set to the sum of the weight of all nodes in the original block. There is an edge between two nodes u' and v' in the contracted graph if the two corresponding blocks in the clustering are adjacent to each other in G, i.e. block u' and block v' are connected by at least one edge. The weight of an edge (u', v') is set to the sum of the weight of edges that run between block u' and block v' of the clustering.

Initial Layout. The process of computing a size-constrained clustering and contracting it is repeated recursively. Then an initial layout is drawn, meaning that each of the two nodes of the coarsest graph is assigned to a position. We place the vertices such that the distance is optimal. The optimal distance of the two vertices is defined and motivated in the next section.

3.2   Uncoarsening and Local Improvement

When the initial layout has been computed, the solution is successively prolongated to the next finer level, where a local maxent-stress minimizer is used to improve the layout. For undoing the contraction, nodes that have been in a cluster are drawn at a random position around the location of its coarse representative. More precisely, let v be a (fine) vertex that is represented by the coarse supervertex v' at $P = (x, y)$. We place v at a random position in a circle around P with radius $r := \sqrt{c(v')}$. We do this by picking an angle uniformly at

random in $[0, 2\pi]$ and a distance to P uniformly at random in $[0, r]$. These two values are then used as a polar coordinate for v with respect to the origin P.

Local Improvement. Our local improvement tries to minimize the maxent-stress on each level of the hierarchy based on Eq. (2). Note, however, that simply iterating Eq. (2) on each level is not sensible since coarse vertices represent a multitude of vertices. These vertices need space to be drawn on the next finer level. Now let u and v be two vertices on the same fixed level. We adjust distances d_{uv} on the current level in the hierarchy under consideration to $\sqrt{c(u)} + \sqrt{c(v)}$ with the intuition that vertices represented by u should be drawn in a circle around u with radius $\sqrt{c(u)}$ (similarly for v).

As Gansner et al. [12], we adjust the value of α in Eq. (2) during the process. Since we want to approximate the maxent-stress, the value should be small. However, it cannot be too small initially since one would only solve a sparse stress model in this case. Hence, following Gansner et al. [12], we set α to one initially and gradually reduce it by $\alpha := 0.3 \cdot \alpha$ until $\alpha_{\min} = 0.008$ is reached.

We call a single update step of the coordinates of all vertices using Eq. (2) an *iteration.* Multiple iterations with the same value of α are called *round.* The current iteration uses the coordinates that have been computed in the previous iteration. We perform at most a iterations with the same value of α in one round. Then we reduce α as described above. If the relative change $||x^{\ell+1} - x^\ell||/||x^\ell||$ in the layout is smaller than some threshold ϵ, we directly reduce the value of α and continue with the next round.

Faster Local Improvement. The local optimization algorithm presented above has a theoretical running time of $\mathcal{O}(n^2)$ per iteration. To speed this up, one can use approximations for the distances in the entropy term in Eq. (2). We do this by taking the cluster structure computed during coarsening into account: Let $V_1 \cup \ldots \cup V_k$ be the corresponding clustering and $M : V \to V' = \{1, \ldots, k\}$ be the mapping that maps a node $v \in V$ to its coarse representative. The first term in Eq. (2) is computed as before and the second term is approximated by using the coordinates of the corresponding coarse vertex. As formula the second term written without the multiplicative factor $\frac{\alpha}{\rho_u}$ becomes

$$\sum_{\substack{u \neq v \\ M(u)=M(v)}} r(u, v) + \sum_{\substack{v' \in V' \\ v' \neq M(u)}} \nu(v') \frac{x_u - x'_{v'}}{\|x_u - x'_{v'}\|^2} - \sum_{\{u,v\} \in E} r(u, v), \qquad (3)$$

where x' maps a coarse vertex to its coordinates and $\nu(v')$ is the *number of nodes* that the coarse vertex represents on the *current* finer level. Note that this is different from the vertex weight $c(v')$ which represents the number of nodes that the coarse vertex represents on the *finest* level. Roughly speaking, we reduced the necessary amount of computation to add up the values of r by summing up the correct values of r for all vertices that are in a sense *close* and using approximations for vertices that are far away. In our context, a vertex is close if it is in the same cluster as the currently processed vertex. If a vertex is not close,

we use the coordinate of its coarse representative instead. We avoid unnecessary computation by scaling the approximated value of r with the number $\nu(v')$ of vertices it represents and adding approximated value of r only once. The last term in Eq. (3) subtracts values of r for $\{u, v\} \in E$ that have been added in good faith in the first two summations.

Note that if M is the identity, then the term in Eq. (3) is the same as in the original Eq. 2. In this case the first two summations add up the r-values for all pairs of vertices and the last sum subtracts the r-values for pairs that are in E.

After the update of the vertices on the current level, we update the coordinates of the vertices on the coarser level used for approximation. We set the coordinate of a vertex v' on the coarser level to the barycenter of the vertices represented by v'.

Note that one obtains even faster algorithms by using a coarser version of the graph that is *multiple levels beneath* the current level in the graph hierarchy. That means instead of using the next coarser graph, we use the contracted graph which is $h > 1$ levels beneath the current graph in the hierarchy – if there is such. Otherwise, we use the coarsest graph in the hierarchy. Obviously this yields a trade-off between solution quality and running time. Also note that this introduces an additional error. To see this, let the coarser vertices that have the same coarse representative on the level used for approximating values of r be called $\mathcal{M}$-vertices (merged vertices). Now, for a vertex on the current level, the r-values of $\mathcal{M}$-vertices are not accounted for in Eq. (3). Hence, we look at the parameter h carefully in Sect. 4 and evaluate its impact on running time and solution quality. We call our algorithms MulMent and denote by MulMent$_h$ the algorithm that uses an h-level approximation of the r-values. With $h = 0$ we denote the quadratic-time algorithm. A rough analysis in TR [27] yields:

Proposition 1. *Under the assumption of equal cluster sizes, the running time of one iteration of algorithm MulMent$_h$, $h \geq 0$, is $\mathcal{O}(m + n^{\frac{h+2}{h+1}})$, respectively.*

Properly implemented, multilevel algorithms lead to fast convergence of their local optimizers. Moreover, the overall work performed by the multilevel approach is only a constant factor times the one on the finest level. This leads us to the initial appraisal that the same asymptotic running times may hold for the respective complete algorithms.

Shared-Memory Parallelization. Our shared-memory parallelization of an iteration of the local optimizer uses OpenMP and works as follows: Since new coordinates of the vertices in the same iteration can be computed independently, we use multiple threads to do so. The relative change in the layout $||x^{\ell+1} - x^\ell|| / ||x^\ell||$ can be computed in parallel using a reduce operation. Parallelism is also used analogously when working on different levels for the distance approximations in the entropy term. Other parts of the overall algorithm could potentially be parallelized, too – such as coarsening. However, already on medium sized graphs coarsening consumes less than 5 % of the algorithm's overall running time. Moreover, the relative running time of coarsening decreases even more with increasing graph size so that the effort does not seem worth it.

4   Experimental Evaluation

Methodology. We implemented[2] the algorithm described above using C++. Parallelization of our algorithm has been done using OpenMP. We compiled our programs using g++ 4.9 -O3 and OpenMP 3.1. Executables for Pivot-MDS (PMDS) [5] and MaxEnt (GHN, for clarity we use the author names as acronym) [12] have been kindly provided by Yifan Hu. When comparing layouts computed by different algorithms, we evaluate two metrics. The first metric is the full stress measure, $F(x) = \sum_{u,v \in V} w_{uv}(||x_u - x_v|| - d_{uv})^2$, and the second one is the maxent-stress function $M(x)$ as defined in Eq. (1) at the final penalty level of $\alpha = 0.008$. The latter is of primary importance since that is what GHN and Mul-Ment optimize for. The implementations PMDS and GHN sometimes compute vertices that are on the same position. Hence, we add small random noise to the coordinates of these layouts in order to be able to compute the maxent-stress. More precisely, for each of the components of the 2D-coordinate of a node, we randomly add or subtract a random value from the interval $[10^{-7}, 10^{-4}]$. This changes the full stress measure by less than 10^{-4} percent on average. We follow the methodology of Gansner et al. [12] and scale the layout of all algorithms to minimize the stress to be fair to all methods: We find a scalar s such that $\sum_{u,v \in V} w_{uv}(s||x_u - x_v|| - d_{uv})^2$ is minimized for a given layout x.

Machine. Our machine has four Octa-Core Intel Xeon E5-4640 (Sandy Bridge) processors (32 cores, 64 with hyperthreading active) which run at a clock speed of 2.4 GHz. It has 512 GB local memory, 20 MB L3-Cache and 8x256 KB L2-Cache. Unless otherwise mentioned, our algorithms use all 64 cores (hyperthreading) of that machine. Since PMDS and GHN are sequential algorithms, they use one core of that machine.

Algorithm Configuration. After an extensive evaluation of the parameters, we fixed the cluster coarsening parameters f to 20 and b to 2. The initial value of the penalty parameter α is set to 1. We perform at most $a = 2$ iterations with the same value of α, while it has not reached its minimum value of 0.008. When it has reached its minimum value, we iterate until the relative error $||x^{\ell+1} - x^{\ell}||/||x^{\ell}||$ is smaller than 0.0001. Yet, our experiments indicate that our algorithm is not very sensitive about the choice of these parameters. We evaluate the influence of the approximation level h in Sect. 4.1.

Instances. We use the instances 1138_bus, USpowerGrid, bcsstk31, commanche and luxembourg employed in [12] and extend the set to include larger instances. We excluded the graphs gd, qh882 and lp_ship04l from [12] from our experiments since the graphs are either not undirected or the corresponding matrix is rectangular. Most of the instances taken from [12] are available at the Florida Sparse

---

[2] We released the implementation of our algorithms as open source in the KaDraw (Karlsruhe Graph Drawing) framework available at http://algo2.iti.kit.edu/kadraw/.

Matrix Collection [6]. The graphs 3elt, bcsstk31, fe_pwt and auto are available at the Walshaw benchmark archive [31]. The graphs del$X$ are Delaunay triangulations of 2^X random points in the unit square [17]. Moreover, the graphs nyc and luxembourg are road networks. These graphs have been taken from the benchmark set of the 9th and 10th DIMACS Implementation Challenge [2,7]. A summary of the basic properties of these instances can be found in the technical report version of this paper [27]. In any case, we draw the largest connected component if the graph has more than one. We assume unit length distance for all graphs.

4.1   Influence of Coarse Graph Approximation and Scalability

In this section, we investigate the influence of the parameter h on layout quality and running time (algorithmic speedup) as well as the scalability of our algorithms with varying number of threads (parallel speedup). We perform detailed experiments on our medium sized networks (using 64 threads) and present parallel speedups on the largest graphs auto and del20. We report absolute running times and parallel speedups for the graph del20 in Fig. 2 and present detailed data for the medium size networks as well as more plots in [27]. We do not report layout quality metrics for auto and del20 since the size of the network makes it infeasible to compute them and the result of the algorithm is independent of the number of threads used.

We now investigate the influence of the parameter h. In general, the larger the graphs get, the larger the algorithmic speedups obtained with increasing h. On the smallest graph in this collection, we obtain an algorithmic speedup of about 3 with $h = 6$ (fe_pwt) over MulMent$_0$. On the largest two instances in this section, we obtain an algorithmic speedup of 30 with $h = 9$ (auto) and of 122 with $h = 10$ (del20). In addition, the precise choice of the parameter does not seem to have a very large impact on solution quality on these graphs. This is also due to the size of the networks. The graphs on which full stress measure slightly increases are luxembourg and bcsstk31 (7 % and 15 % respectively – see [27]). The metric actually under consideration, maxent-stress, always remains comparable. On all instances under consideration, we observe a locally optimal value for h in terms of running time. It is around seven and seems to get larger with increasing graph size. This is due to the fact that too large values of h provide less precision and slower convergence.

On del20, the scalability with the number of threads is almost perfect for small values of h. With enabled hyperthreading, we achieve slightly *superlinear* speedups for MulMent$_0$. As less work has to be done for increasing h, speedups get smaller. The smallest speedup on this graph has been observed for MulMent$_{10}$. In this case, we achieve a speedup of 11.5 using 64 threads over MulMent$_{10}$ using one thread. With even larger h speedups increase again. The parallel scalability on auto is similar.

Another interesting way to look at the data is the overall speedup – algorithmic and parallel speedup combined – achieved over MulMent$_0$ using only one thread. The largest overall speedup is obtained by MulMent$_{10}$ using 64 threads.
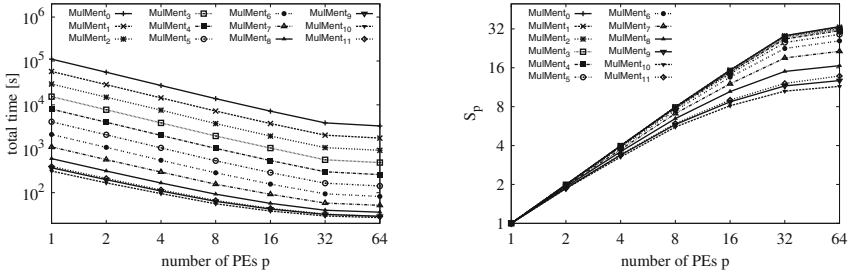
Fig. 2. Running times and parallel speedups of our algorithms on del20.

In this case, the overall speedup is larger than 4000 – reducing the running time of the algorithm from 30 hours to 27 seconds. Speedups over PMDS and GHN are found in the next section.

4.2   Comparison to Other Drawing Algorithms

We now compare MulMent to the two implementations PMDS [5] and GHN [12]. We do this on all networks but only report quality metrics for small and medium sized graphs since it is infeasible to compute quality metrics for the large graphs. We report detailed data in [27].

Most importantly, although MulMent sometimes performs a few percent worse than GHN, the maxent-stress of all layouts is more or less *similar*. PMDS performs slightly worse in this metric. Intriguingly, the alternative full stress metric is consistently better on small networks for MulMent than the results obtained by PMDS (except for $h = 10$). On the other hand, full stress obtained by our algorithms is comparable to the layout computed by GHN on four out of nine instances. On the three largest medium sized networks, we obtain worse full stress than PMDS and GHN. However, this is not astonishing since our algorithm does not optimize for full stress – in *contrast* to PMDS. And GHN at least starts with a PMDS solution and improves maxent-stress afterwards.

Our implementations of $\text{MulMent}_{7,10}$ are always faster than GHN, both of them a factor 30 on average. Also, $\text{MulMent}_{7,10}$ outperform even PMDS in terms of running time as soon as the graphs get large enough (medium and large sized graphs). On the large graphs, MulMent_{10} is a factor of 2 to 3 faster than PMDS and a factor of 32 to 63 faster than GHN. In addition, $\text{MulMent}_{7,10}$ are also several times faster than GHN when using one thread only (see TR [27]).

4.3   Dynamic Networks

One of the main advantages of the iterative scheme is its ability to use an existing layout for computing a new one, e. g. for a graph that has changed over time. We perform experiments with dynamic graphs obtained by modifying our medium sized networks. Often one is interested in drawing graphs that have more or less good locality. Hence, we define a random model that modifies the edges of a

graph by removing random edges and inserting edges between vertices that are not too far apart.

To be more precise, we start with an input graph G and perform a breadth first search from a random start node to compute a random spanning tree. We then remove $x\%$ undirected non-tree edges at random in the beginning. Note that this ensures that the graph stays connected. Afterwards, we insert $x\%$ new edges as follows. We pick a random node and insert an undirected edge to a random node that has distance $1 < d \leq \mathcal{D}$ in the original graph G, where $\mathcal{D}$ is a tuning parameter. We denote the graph that results out of this process as Q.

We compute two layouts of Q. The first one updates coordinates given by an initial layout of G (update algorithm). The second layout is computed by our algorithm from scratch (scratch algorithm), i. e. discarding the initial layout. In the first case, we start directly at the penalty level $\alpha = 0.008$ and only update coordinates on the finest level of the hierarchy. We compute the graph hierarchy as before but stop the coarsening process after the computation of h levels. Coordinates of the vertices on the approximation level are set to the middle point of the vertices in the corresponding cluster initially.

We vary $x \in \{1, 5\}$, $\mathcal{D} \in \{2, 16\}$ and $h \in \{0, 7\}$, and present detailed data in [27]. As expected, the *running time* of the update algorithm (t_{dyn}) is always smaller than the running time of the scratch algorithm (t_{scratch}). As MulMent$_7$ performs less work than MulMent$_0$, algorithmic speedups are always larger for the latter. For $h = 0$, the update algorithm is a factor of 4 faster than the scratch algorithm on average. On the other hand, for $h = 7$ the update algorithm saves about 50% time on average over the scratch algorithm. *Solution quality* is not influenced much. On average, the full stress measure of the update algorithm is 9% larger and maxent-stress improves by 1% compared to the scratch algorithm. The increase in full stress is mostly due to the Delaunay instance and $\mathcal{D} = 16$, in which the full stress of the layout of the update algorithm is a factor of two larger. The algorithmic speedup does not seem to be largely influenced by $\mathcal{D}$. However, we expect that much larger values of $\mathcal{D}$ will decrease the speedup of the update algorithm over the scratch algorithm.

5   Conclusions

We have presented a new multilevel algorithm for iteratively and approximatively optimizing the maxent-stress model, a model proposed by Gansner et al. [12] to avoid typical pitfalls of other stress models. From the experimental evaluation we conclude that our parallel algorithm produces layouts with similar visual quality and maxent-stress values as the reference implementation [12]. At the same time it is on average 30 times faster, even more for dynamic graphs. Moreover, our algorithm is even up to twice as fast as the fastest stress-based algorithm PivotMDS [5]. It thus combines the high speed of PivotMDS with the high visual quality of Maxent in a single algorithm, at least if a multicore system is available.

Currently our method is only capable of handling constant edge lengths. This requirement is due to the way coarse vertices are placed and later interpolated to a finer level. In future work we would like to eliminate this limitation.

References

1. Abello, J., van Ham, F., Krishnan, N.: ASK-GraphView: a large scale graph visualization system. IEEE Trans. Vis. Comput. Graph. **12**(5), 669–676 (2006)
2. Bader, D.A., Meyerhenke, H., Sanders, P., Schulz, C., Kappes, A., Wagner, D.: A benchmark set for graph clustering and graph partitioning. In: Encyclopedia of Social Network Analysis and Mining (2014)
3. Barnes, J., Hut, P.: A hierarchical $O(n \log n)$ force-calculation algorithm. Nature **324**, 446–449 (1986)
4. Bartel, G., Gutwenger, C., Klein, K., Mutzel, P.: An experimental evaluation of multilevel layout methods. In: Brandes, U., Cornelsen, S. (eds.) GD 2010. LNCS, vol. 6502, pp. 80–91. Springer, Heidelberg (2011)
5. Brandes, U., Pich, C.: Eigensolver methods for progressive multidimensional scaling of large data. In: Kaufmann, M., Wagner, D. (eds.) GD 2006. LNCS, vol. 4372, pp. 42–53. Springer, Heidelberg (2007)
6. Davis, T.: The university of florida sparse matrix collection (2008). http://www.cise.ufl.edu/research/sparse/matrices
7. Demetrescu, C., Goldberg, A.V., Johnson, D.S.: The Shortest Path Problem: 9th DIMACS Implementation Challenge, vol. 74. AMS (2009)
8. Eades, P.: A heuristic for graph drawing. Congressus Numerantium **42**, 146–160 (1984)
9. Frishman, Y., Tal, A.: Multi-level graph layout on the GPU. IEEE Trans. Vis. Comput. Graph. **13**(6), 1310–1319 (2007)
10. Fruchterman, T.M.J., Reingold, E.M.: Graph drawing by force-directed placement. Softw. Pract. Exp. **21**(11), 1129–1164 (1991)
11. Gajer, P., Kobourov, S.G.: GRIP: Graph drawing with intelligent placement. J. Graph Algorithms Appl. **6**(3), 202–224 (2002)
12. Gansner, E.R., Hu, Y., North, S.C.: A maxent-stress model for graph layout. IEEE Trans. Vis. Comput. Graph. **19**(6), 927–940 (2013)
13. Gansner, E.R., Koren, Y., North, S.C.: Graph drawing by stress majorization. In: Pach, J. (ed.) GD 2004. LNCS, vol. 3383, pp. 239–250. Springer, Heidelberg (2005)
14. Godiyal, A., Hoberock, J., Garland, M., Hart, J.C.: Rapid multipole graph drawing on the GPU. In: Tollis, I.G., Patrignani, M. (eds.) GD 2008. LNCS, vol. 5417, pp. 90–101. Springer, Heidelberg (2009)
15. Hachul, S., Jünger, M.: Large-graph layout algorithms at work: an experimental study. J. Graph Algorithms Appl. **11**(2), 345–369 (2007)
16. Hachul, S., Jünger, M.: Drawing large graphs with a potential-field-based multilevel algorithm. In: Pach, J. (ed.) GD 2004. LNCS, vol. 3383, pp. 285–295. Springer, Heidelberg (2005)
17. Holtgrewe, M., Sanders, P., Schulz, C.: Engineering a scalable high quality graph partitioner. In: IEEE Parallel and Distributed Computing (IPDPS 2010), pp. 1–12. IEEE (2010)
18. Hoske, D., Lukarski, D., Meyerhenke, H., Wegner, M.: Is nearly-linear the same in theory and practice? a case study with a combinatorial laplacian solver. In: Bampis, E. (ed.) SEA 2015. LNCS, vol. 9125, pp. 205–218. Springer, Heidelberg (2015)

19. Hu, Y., Shi, L.: Visualizing large graphs. Wiley Interdisc. Rev. Comput. Stat. **7**(2), 115–136 (2015)
20. Ingram, S., Munzner, T., Olano, M.: Glimmer: multilevel MDS on the GPU. IEEE Trans. Vis. Comput. Graph. **15**(2), 249–261 (2009)
21. Kamada, T., Kawai, S.: An algorithm for drawing general undirected graphs. Inf. Process. Lett. **31**, 7–15 (1989)
22. Kirmani, S., Raghavan, P.: Scalable parallel graph partitioning. In: High Performance Computing, Networking, Storage and Analysis (S 2013), pp. 51:1–51:10. ACM (2013)
23. Kobourov, S.G.: Force-directed drawing algorithms. In: Tamassia, R. (ed.) Handbook of Graph Drawing and Visualization, Chap. 12, pp. 383–408. CRC Press, Boca Raton (2013)
24. Koutis, I., Miller, G.L., Tolliver, D.: Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing. Comput. Vis. Image Underst. **115**(12), 1638–1646 (2011)
25. Kruskal, J.B.: Multidimensional scaling by optimizing goodness of fit to a non-metric hypothesis. Psychometrika **29**(1), 1–27 (1964)
26. Livne, O.E., Brandt, A.: Lean algebraic multigrid (LAMG): fast graph Laplacian linear solver. SIAM J. Sci. Comput. **34**(4), B499–B522 (2012)
27. Meyerhenke, H., Nöllenburg, M., Schulz, C.: Drawing large graphs by multilevel maxent-stress optimization. CoRR, arXiv:1506.04383 (2015)
28. Meyerhenke, H., Sanders, P., Schulz, C.: Partitioning complex networks via size-constrained clustering. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 351–363. Springer, Heidelberg (2014)
29. Quigley, A., Eades, P.: FADE: graph drawing, clustering, and visual abstraction. In: Marks, J. (ed.) GD 2000. LNCS, vol. 1984, pp. 197–210. Springer, Heidelberg (2001)
30. Raghavan, U.N., Albert, R., Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Phy. Rev. E **76**(3), 036106 (2007)
31. Soper, A.J., Walshaw, C., Cross, M.: A combined evolutionary search and multi-level optimisation approach to graph-partitioning. J. Global Optim. **29**(2), 225–241 (2004)
32. Walshaw, C.: A multilevel algorithm for force-directed graph-drawing. J. Graph Algorithms Appl. **7**(3), 253–285 (2003)