Faster Force-Directed Graph Drawing with the Well-Separated Pair Decomposition

Fabian Lipp(✉), Alexander Wolff, and Johannes Zink

Lehrstuhl für Informatik I, Universität Würzburg, Würzburg, Germany
fabian.lipp@uni-wuerzburg.de
http://www1.informatik.uni-wuerzburg.de/en/staff

Abstract. The force-directed paradigm is one of the few generic approaches to drawing graphs. Since force-directed algorithms can be extended easily, they are used frequently. Most of these algorithms are, however, quite slow on large graphs as they compute a quadratic number of forces in each iteration. We speed up this computation by using an approximation based on the well-separated pair decomposition.

We perform experiments on a large number of graphs and show that we can strongly reduce the runtime—even on graphs with less then a hundred vertices—without a significant influence on the quality of the drawings (in terms of number of crossings and deviation in edge lengths).

1 Introduction

Force-directed algorithms are commonly used to draw graphs. They can be used on a wide range of graphs without further knowledge of the graphs' structure. The idea is to define physical forces between the vertices of the graph. These forces are applied to the vertices iteratively until stable positions are reached. The well-known spring-embedder algorithm of Eades [5] models the edges as springs. His approach was refined by Fruchterman and Reingold [9]. Between pairs of adjacent vertices they apply attracting forces caused by springs. To prevent vertices getting too close, they apply repulsive forces between all pairs of vertices.

Generally, force-directed methods are easy to implement and can be extended well. For example, Fink et al. [7] defined additional forces to draw Metro lines in Metro maps as Bézier curves instead of as polygonal chains. Different aesthetic criteria can be balanced by weighing them accordingly. Force-directed algorithms can in principle be used for relatively large graphs with hundreds of vertices and often yield acceptable results. Unfortunately, force-directed methods are rather slow on such graphs. This is caused by the computation of the repulsive force for every vertex pair, which yields a quadratic runtime for each iteration. In this paper, we present a new approach to speed this up.

Previous Work. There are a lot of techniques to speed up force-directed algorithms. For example, Barnes and Hut [1] use a quadtree, a multi-purpose spatial

E. Di Giacomo and A. Lubiw (Eds.): GD 2015, LNCS 9411, pp. 52–59, 2015.
DOI: 10.1007/978-3-319-27261-0_5

data structure, to approximate the forces between the vertex pairs. We will compare our algorithm to theirs subsequently. Another approach is the multilevel paradigm introduced by Walshaw [15]. After contracting dense subgraphs, the resulting coarse graph is laid out. Then the vertices are uncontracted and a layout of the whole graph based on the coarse layout is computed. This can be done over several levels. The multilevel paradigm does not rule out our WSPD-based approach; our approach can be applied to each level.

Various force-directed graph drawing algorithms have been compared before [2,3,8]. We use some of the quality criteria described in the literature to evaluate our algorithm.

Callahan and Kosaraju [4] defined a decomposition for point sets in the plane, the *well-separated pair decomposition* (WSPD). Given a point set P and a number $s > 0$, this decomposition consists of pairs of subsets $(A_i, B_i)_{i=1,\ldots,k}$ of P with two properties. First, for each pair $(p, q) \in P^2$ with $p \neq q$, there is a unique index $i \in \{1, \ldots, k\}$ such that $p \in A_i$ and $q \in B_i$ or vice versa. Second, each pair (A_i, B_i) must be s-well-separated, that is, the distance between the two sets is at least s times the larger of the diameters of the sets. Callahan and Kosaraju showed how to construct a WSPD for a set of n points in $O(n \log n)$ time where the number k of pairs of sets is linear in n.

The WSPD has been used for graph drawing before; Gronemann [10] employed it to speed up the fast multipole multilevel method [11]. While our WSPD is based on the *split tree* [4], Gronemann's is based on a quadtree.

Our Contribution. We use the WSPD in order to speed up the force-directed algorithm of Fruchterman and Reingold (FR). Instead of computing the repulsive forces for every pair of points, we represent every set $A_1, \ldots, A_k, B_1, \ldots, B_k$ in the decomposition by its barycenter and use the barycenter of a set, say A_i, as an approximation when computing the forces between this set and a point in B_i. Thus, an iteration takes us $O(n \log n)$ time, instead of $\Omega(n^2)$ for the classical algorithm.

Additionally, our method is very simple and allows the user to define forces arbitrarily—as long as the total force on a point p is the sum of the forces of point pairs in which p is involved. Hence, our approach can be applied to other force-directed algorithms as well. We don't consider other techniques such as Multidimensional Scaling (MDS) or multi-level algorithms in this paper, as we only want to show that we can speed up a force-directed graph layout algorithm using the WSPD. We guess that this technique can be applied to other algorithms as well. In the above-mentioned fast multipole method, in contrast, the approximation of the repulsive forces is quite complicated (as Hachul and Jünger [11] point out); it requires the expansion of a Laurent series.

2 Algorithm

In this section, we describe our WSPD-based implementation, analyze its asymptotic running time, and give a heuristic speed-up method.

Constructing a WSPD. There are various ways to construct an *efficient* WSPD, that is, a WSPD with a linear number of pairs of sets. We use the *split tree* as described by Callahan and Kosaraju [4] when introducing the WSPD. Our implementation follows the algorithm FastSplitTree in the textbook of Narasimhan and Smid [13, Sect. 9.3.2]. Given n points, this algorithm constructs a linear-size split tree in $O(n \log n)$ time. Given the tree, a WSPD with separation constant s can be built in $O(s^2 n)$ time.

The Force-Directed Algorithm. The general principle of a force-directed algorithm is as follows. In every iteration, the algorithm computes forces on the vertices. These forces depend on the current position of the vertices in the drawing. The forces are applied as an offset to the position of each vertex. The algorithm terminates after a given number of iterations or when the forces get below a certain threshold.

A classical force-directed algorithm such as FR computes, in every iteration, an attractive force for any pair of adjacent vertices and a repulsive force for any pair of vertices. Fruchterman and Reingold [9] use $F_{\text{attractive}}(u, v) = d^2/c$ and $F_{\text{repulsive}}(u, v) = -c^2/d$, where c is a constant describing the ideal edge length and $d = d(u, v)$ is the distance between vertices u and v in the current drawing.

Our modified algorithm is shown in Algorithm 1. We first compute a fair split tree T for the current positions of the vertices of G (which are stored in the leaves of T). Each node μ of T corresponds to the set of vertices in the leaves of the subtree rooted in μ. Bottom-up, we compute the barycenters of the sets corresponding to the nodes of T. From T, we compute a WSPD $(A_i, B_i)_i$ for the current vertex positions. Each set A_i (and B_i) of the WSPD corresponds to a node α_i (and β_i) of T. For each pair (A_i, B_i) of the WSPD, we compute $F_{\text{repulsive}}$ from the barycenter of A_i to the barycenter of B_i (and vice versa), and store the results (in an accumulative fashion) in α_i and β_i. Finally, we traverse T top-down. During the traversal, we add to the force of each node the force of the parent node. When we reach the leaves of T, which correspond to the graph vertices, we have computed the resulting force for each vertex.

Running Time. We denote the number of vertices of the given graph by n and the number of edges by m. In each iteration, the classical algorithm computes the attractive forces in $O(m)$ time and the repulsive forces in $O(n^2)$ time.

We don't modify the computation of the attractive forces. For computing the repulsive forces, the most expensive step is the computation of the split tree T and the WSPD, which takes $O(n \log n)$ time. The barycenters of the sets corresponding to the nodes of T can be computed bottom-up in linear time. The forces between the pairs of the WSPD can also be computed in linear total time. The same holds for the forces acting on the vertices. In total, hence, an iteration takes $O(m + n \log n)$ time.

Improvements. To speed up our algorithm, we compute a new split tree and the resulting WSPD only every few iterations. To be precise, we only recompute it when $\lfloor 5 \log i \rfloor$ changes, where i is the current iteration. Thus, the WSPD may

Algorithm 1. WSPD-based force computation for a graph $G = (V, E)$

```
// attractive forces for adjacent vertices:
```
foreach $e = (u, v) \in E$ **do**
  u.addForce($F_{\text{attractive}}(u, v)$); v.addForce($F_{\text{attractive}}(v, u)$)

```
// approximation of repulsive forces:
```
Compute a fair split tree T for the current positions of the vertices (stored in the leaves of T).
For each node μ of T, compute the barycenter $c(\mu)$ of the leaves of the subtree rooted in μ.
Compute a WSPD $(A_i, B_i)_{i=1,\ldots,k}$ from T; each A_i (B_i) corresponds to a node α_i (β_i) of T.
for $i = 1$ **to** k **do**
  α_i.addForce($|B_i| \cdot F_{\text{repulsive}}(c(\alpha_i), c(\beta_i))$)
  β_i.addForce($|A_i| \cdot F_{\text{repulsive}}(c(\beta_i), c(\alpha_i))$)

Traverse the split tree top-down to compute the total force for every vertex of G.

not be valid for the current vertex positions. This makes the approximation of the forces more inaccurate, but our experiments show that this method does not change the quality of the drawings significantly, while the running time decreases notably (see Figs. 1, 2 and 3).

Implementation. Our Java implementation is based on FRLayout, the FR algorithm implemented in the JUNG library [12]. We slightly optimized the code, which reduced the runtime by a constant factor. Additionally, we removed the frame that bounded the drawing area, as it caused ugly drawings for larger graphs. For our experimental comparison in Sect. 3, we used FRLayout with these modifications. It is this implementation that we then sped up using the WSPD. We recompute the WSPD only every few iterations as described in the previous paragraph. We call the result FR+WSPD. For comparison, we also implemented the quadtree-based speed-up method of Barnes and Hut [1], which we call FR+Quad, and a grid-based approach suggested already by Fruchterman and Reingold [9], which we call FR+Grid. To widen the scope of our study, we included some algorithms implemented in C++ in the *Open Graph Drawing Framework* (OGDF, www.ogdf.net): GEM, FM^3 (with and without multilevel technique, then we call it FM^3 single) of Hachul and Jünger [11], and FRExact (the exact FR implementation in OGDF).

3 Experimental Results

We formulate the following hypotheses which we then test experimentally.

(H1) The quality of the drawings produced by FR+WSPD is comparable to that of FRLayout.
(H2) On sufficiently large graphs, FR+WSPD is faster than FRLayout.

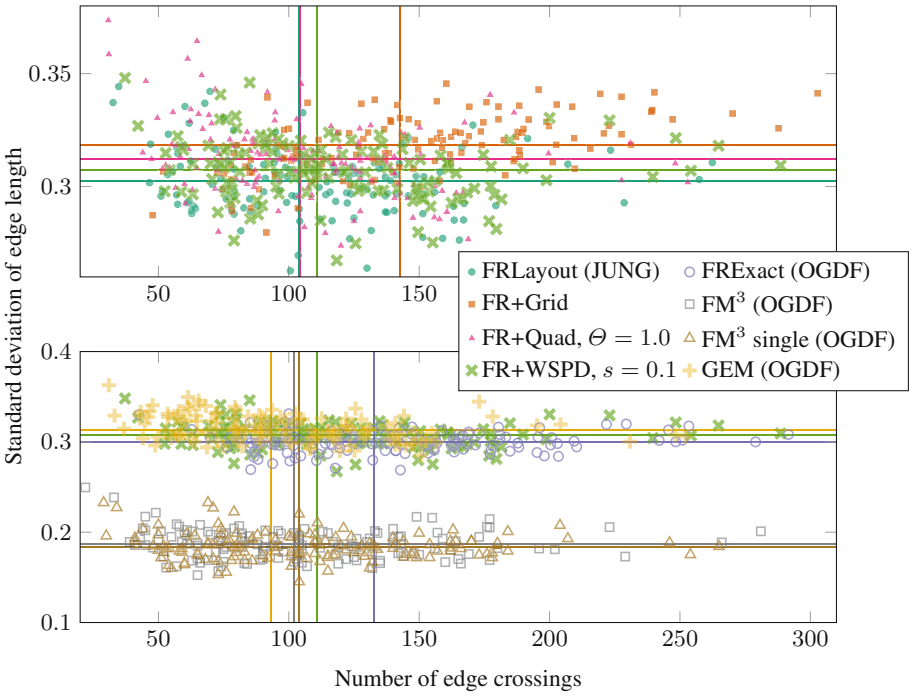

Fig. 1. Standard deviation of the edge length (y-axis) over number of edge crossings (x-axis) for various variants of the algorithm, applied to all 140 Rome graphs with exactly 100 vertices. Top: quality of the unmodified FRLayout algorithm and its variants FR+WSPD, FR+Quad, and FR+Grid. Bottom: FR+WSPD and some algorithms implemented in OGDF. For each algorithm, a vertical and a horizontal line mark its median performance.

We assume these hypotheses due to the favorable properties of the WSPD: the separation property hints at (H1) and the improved time complexity per iteration implies (H2).

We tested our algorithms on two data sets; (i) the Rome graph collection [14] that contains 11528 undirected connected graphs with 10–100 vertices each, and (ii) 40 random graphs that we generated using the EppsteinPowerLawGenerator [6] in JUNG, which yields graphs whose structure is similar to Web graphs. Our graphs had 2,500, 5,000, 7,500, ..., 100,000 vertices and 2.5 times as many edges. We considered only the largest connected component of each generated graph.

The experiments were performed on an Intel Xeon CPU with 2.67 GHz and 20 GB RAM running Linux. The computer has 16 cores, but we did not parallelize our code. During our experiments, only one core was operating at close-to-full capacity.

We measured the quality of the drawings by (a) the number of edge crossings and (b) the standard deviation of the edge length (normalized by the mean edge length). These criteria have been used before to compare force-directed layout algorithms [3,8].

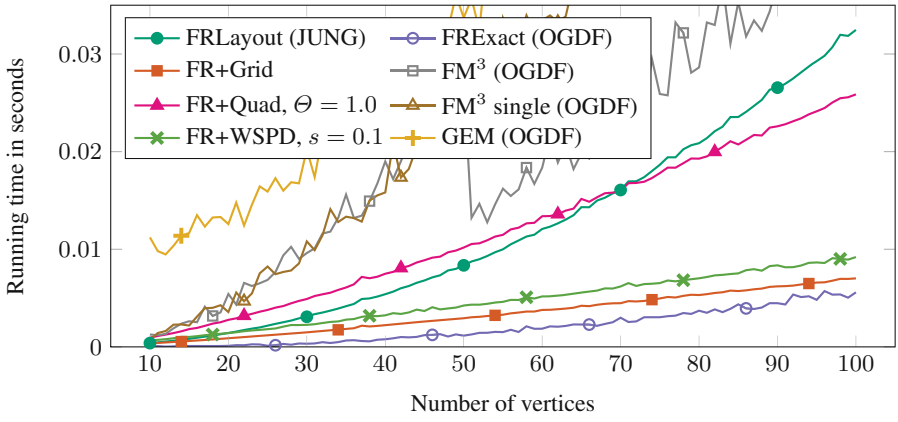

Fig. 2. The runtimes for the different variants of the algorithm as a function of the number of vertices. Each point in the plots represents the mean value of the runtimes on all Rome graphs with the given number of vertices. The markers are used only as a tool to identify the plots.

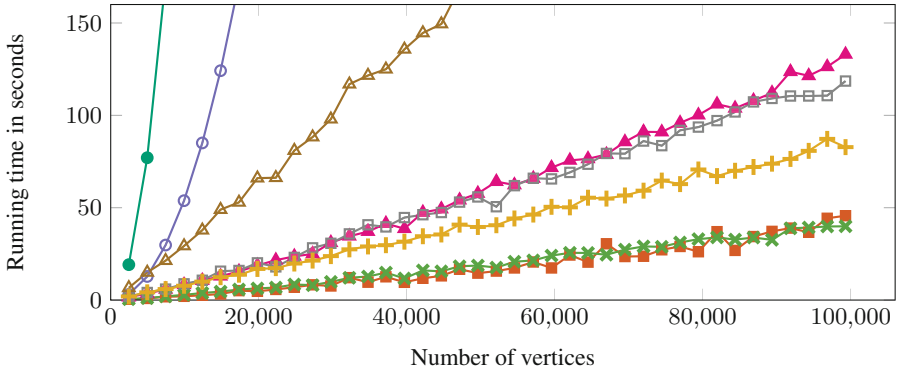

Fig. 3. The runtimes of the algorithms on 40 random graphs. The graphs were generated using the EppsteinPowerLawGenerator [6] in JUNG with $|V| = 2{,}500, 5{,}000, 7{,}500, \ldots, 100{,}000$ and $|E| \approx 2.5 \cdot |V|$. Only the largest connected component of each generated graph was considered. Other than in Fig. 2, each marker represents one of the tested graphs. The legend is the same.

To test hypothesis (H1), we compared the quality of the drawings of FRLayout, FR+WSPD, FR+Quad, and FR+Grid. In order to vary as few parameters as possible, we kept the size of the graphs constant in this part of the study. We used all Rome graphs with exactly 100 vertices. The 140 graphs have, on average, 135 edges.

We first compared the outputs of FR+WSPD for different values (0.01, 0.1, 1) of the separation constant s. The distribution of the results in the plot was roughly the same, that is, the quality of the drawings did not strongly depend on s. Using $s = 1$ was about 30 % slower than $s = 0.1$ or $s = 0.01$.

Similarly, FR+Quad has a parameter Θ that controls how fine the given point set is subdivided. Increasing Θ decreases the running time. Our experiments confirmed what Barnes and Hut [1] observed: only values of Θ close to 1 give results with a similar quality as the unmodified algorithm.

The upper scatterplot in Fig. 1 compares variants of FR based on different speed-up techniques. Compared to FRLayout, FR+Quad is slightly worse in terms of uniformity of edge lengths and FR+WSPD is slightly worse in terms of edge crossings and between FRLayout and FR+Quad in terms of edge lengths. FR+Grid is worse in both measures, especially in the number of edge crossings. Hence, there is support for hypothesis (H1).

The lower scatterplot in Fig. 1 compares FR+WSPD to the above-mentioned OGDF algorithms. In terms of uniformity of edge lengths, there are two clear clusters: the two FM^3 variants are better than the rest. In terms of crossings, GEM is best, followed by the FM^3 variants, and then by FR+WSPD, which surprisingly is *better* than FRExact.

To test hypothesis (H2), we measured the runtimes of the all algorithms on the Rome graphs (Fig. 2) and the random graphs (Fig. 3). In Java, we only measure the time used for the thread running the force-directed algorithm in our Java Virtual Machine; this eliminates the influence of the garbage collector and the JIT compiler on our measurements. In C++, we used an OGDF method for measuring the CPU time. For each graph size, we display the mean runtime over all graphs of that size.

The results are as follows. As expected, FR+WSPD (with $s = 0.1$) is faster than FRLayout on larger graphs. We were surprised, however, to see that FR+WSPD overtakes FRLayout already around $n \approx 30$. FR+WSPD also turned out to be faster than FR+Quad (with $\Theta = 1$) and than FM^3 by a factor of 1.5 to 3. FR+WSPD and FR+Grid are comparable in speed, and twice as fast as GEM. Recall, however, that FR+Grid tends to produce more edge crossings (Fig. 1). Concerning the comparison between Java and C++, FRExact (in C++) is roughly four times faster than FRLayout (in Java).

Conclusion. Our experiments show that the WSPD-based approach speeds up force-directed graph drawing algorithms such as FR considerably without sacrificing the quality of the drawings. The main feature of the new approach is its simplicity. We plan to combine our approach with multi-level techniques in order to draw much larger graphs.

References

1. Barnes, J., Hut, P.: A hierarchical $O(N \log N)$ force-calculation algorithm. Nature **324**(6096), 446–449 (1986)
2. 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)
3. Brandenburg, F.J., Himsolt, M., Rohrer, C.: An experimental comparison of force-directed and randomized graph drawing algorithms. In: Brandenburg, F.J. (ed.) GD 1995. LNCS, vol. 1027, pp. 76–87. Springer, Heidelberg (1996)

4. Callahan, P.B., Kosaraju, S.R.: A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. J. ACM **42**(1), 67–90 (1995)
5. Eades, P.: A heuristics for graph drawing. Congr. Numerantium **42**, 146–160 (1984)
6. Eppstein, D., Wang, J.Y.: A steady state model for graph power laws. In: 2nd International Workshop Web Dynamics (2002). http://arxiv.org/abs/cs/0204001
7. Fink, M., Haverkort, H., Nöllenburg, M., Roberts, M., Schuhmann, J., Wolff, A.: Drawing metro maps using Bézier curves. In: Didimo, W., Patrignani, M. (eds.) GD 2012. LNCS, vol. 7704, pp. 463–474. Springer, Heidelberg (2013)
8. Frick, A., Ludwig, A., Mehldau, H.: A fast adaptive layout algorithm for undirected graphs. In: Tamassia, R., Tollis, I.G. (eds.) GD 1994. LNCS, vol. 894, pp. 388–403. Springer, Heidelberg (1995)
9. Fruchterman, T.M.J., Reingold, E.M.: Graph drawing by force-directed placement. Softw. Pract. Exp. **21**(11), 1129–1164 (1991)
10. Gronemann, M.: Engineering the fast-multipole-multilevel method for multicore and SIMD architectures. Master's thesis, Department of Computer Science, TU Dortmund (2009)
11. 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)
12. JUNG: Java Universal Network/Graph Framework. http://jung.sourceforge.net. Accessed 2 September 2015
13. Narasimhan, G., Smid, M.: Geometric Spanner Networks. Cambridge University Press, New York, NY, USA (2007)
14. Rome Graphs. http://graphdrawing.org/data.html, http://www.graphdrawing.org/download/rome-graphml.tgz. Accessed 2 September 2015
15. Walshaw, C.: A multilevel algorithm for force-directed graph-drawing. J. Graph Algorithms Appl. **7**(3), 253–285 (2003)