Multipole-Based Force Approximation Revisited – A Simple but Fast Implementation Using a Dynamized Enclosing-Circle-Enhanced k-d-Tree

Ulrich Lauther

Siemens AG, CT SE 6
Otto-Hahn Ring 6, 81730 München, Germany
ulrich.lauther@siemens.com

Abstract. Most force-directed graph drawing algorithms depend for speed crucially on efficient methods for approximating repulsive forces between a large number of particles. A combination of various tree data structures and multi-pole approximations has been successfully used by a number of authors. If a multi-level approach is taken, in the late (and due to the large number of particles computationally intensive) steps, movements of particles are quite limited. We utilize this fact by basing force-calculations on an easy updatable tree data structure. Using explicit distance checks instead of relying on implicit guarantees provided by quadtrees and avoiding local expansions of the multi-pole expansion leads to a very simple implementation that is faster than comparable earlier methods. The latter claim is supported by experimental results.

1 Introduction

In several force-directed graph drawing methods [1,2,3,4] the nodes of a graph to be drawn are modeled as charged particles that repel each other and edges are represented as springs with some "ideal" length that attract or repel the two incident nodes, depending on their actual length. A placement of nodes that minimizes the total energy in the system often results in a nice straight line drawing of the underlying graph. This placement is achieved by iteratively computing the total force acting on each node and then moving the nodes accordingly.

In order to make this approach work in practice for large graphs, two considerations are crucial: Firstly, as a naive approach to calculating repulsive forces would take $O(N^2)$ steps per iteration for a graph with N nodes, we need a fast method for approximating these forces with sufficiently high accuracy.

Secondly, to avoid an overly large number of iterations, we need a multilevel approach that first generates a series of coarsened graphs $G_1,\ldots,G_k$ from the initial graph G_0, generates a drawing for the small graph G_k, and then refines this drawing by constructing an initial placement for the nodes of G_i from the node positions in G_{i+1} which is then optimized by force-directed iterations. Note, that at each refinement level we start with a "reasonable" initial placement of nodes.

M. Kaufmann and D. Wagner (Eds.): GD 2006, LNCS 4372, pp. 20–29, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Though we have implemented such a coarsening/refinement approach (along the lines suggested by Hachul and Jünger [5]), it is not further discussed in this paper; its focus is on a simple and efficient approach to approximation of repulsive forces.

2 Previous Work

Beginning with Barnes and Hut [6] a number of force approximation schemes have been developed and used in the context of graph drawing [7,8] that use quadtrees or variants thereof for hierarchical clustering of particles and approximating forces either by substituting a "group particle" at the center of gravity for the particles of a cluster [6] or by calculation of *multipole expansions* [9] for the clusters. These approaches differ in how the tree is constructed and whether explicit distance calculations are used to steer the following force calculations or if implicit properties of the quadtree's topology are used. Hachul [10] , after careful analysis of different quadtree construction methods, came up with the *reduced bucket quadtree* that can be constructed in $O(N \log N)$ steps for N particles.

One drawback of quadtrees is that they divide the area containing the particles into subregions of equal size, not equal population. If particles are not uniformly distributed, they become unbalanced and quite involved algorithms [10] are needed to achieve $O(N \log N)$ complexity for tree construction.

3 The Enclosing-Circle-Enhanced Modified k-d-Tree

As opposed to quadtrees, k-d-trees introduced by Bentley [11] partition the set of particles based on population, not on space.

3.1 Definition of the Tree Data Structure

We use a variant of a 2-d-tree, defined as follows: The root node represents the set of all N particles. Each node representing k particles has two children, where the left child contains the $k/2$ particles with lowest coordinates and the right child the remaining $k - k/2$ particles. (Note: throughout the paper we use integer division). If the bounding rectangle of a node with dimensions dx and dy has $dx > dy$, i.e., it is "horizontally oriented", the x-coordinate is used for splitting, otherwise the y-coordinate. (This is different from Bentley's k-d-trees, where coordinate directions are used in a strict round robin fashion; it helps to avoid long skinny regions which group points together that are far away from each other).

The leafs of the tree are buckets, i.e., they contain a list of particles. The bucket size is bounded by a predefined parameter B. Each bucket contains b particles with $B/2 \leq b \leq B$. (Unless pathologically $N < B/2$).

In each leaf we store center and radius of the smallest circle enclosing all particles of its bucket. The interior nodes too contain enclosing circles of their

subtree, either exact values or upper bounds, depending on the tree construction method, see below.

3.2 Tree Construction

Such a tree can be constructed in $O(N \log N)$ steps as follows:

First, we build two singly linked lists of particles and sort one list by x-coordinates, the other one by y-coordinates. This takes $O(N \log N)$ steps using list merge sort or any other appropriate method.

We create the root node, determine its appropriate splitting direction and append the node, its splitting direction, and the two initial lists to a queue of nodes to be processed.

Then, in the main loop of the algorithms, a node, its lists, and its splitting direction are popped from the queue and the node is split if it contains more than B particles. Splitting a node proceeds as follows: we traverse one of the two sorted lists up to the median element, mark the traversed elements, and transfer them to a list $L1$ (this takes $k/2$ steps for a node representing k particles). The remaining particles are transferred to list $L2$ (list concatanation, one step). Next, we split the second list (the other coordinate direction) based on the marks just made: elements are popped from the list and depending on their mark appended to one or the other of two result lists. Note, that the lists stay sorted during this step. This costs another k steps. Finally, two child nodes are created and appended to the queue together with the four lists just created. If the node taken from the queue contains not more than B nodes, a leaf is created containing one of the associated particle lists. By construction the resulting tree is fully balanced. The work at each level of the tree is $O(N)$. Thus, with $O(\log N)$ tree levels and including the initial sorting step, the overall complexity is $O(N \log N)$, independent of how particles are distributed in the plane.

It remains to discuss how the enclosing circles are created. For the leafs, we could employ any algorithm without affecting the overall complexity, as the size of the buckets is bounded. However, luckily, there is a very efficient method that computes the smallest enclosing circle for a set of n points in the plane in $O(n)$ expected time, introduced by Welzl [12]. If we used this method during tree construction, when the sets of particles for each node and leaf are readily available, the overall *expected* time complexity became $O(N \log N)$.

However, there is a faster method - both from the complexity point of view and in practice -, if we sacrifice some accuracy. We proceed in four passes: In the first pass, the tree is constructed as described. In the second pass, smallest enclosing circles are calculated for leafs, using Welzl's algorithm. In the third pass, traversing the tree from the leafs bottom up, the bounding circles of nodes are initialized with the largest leaf-circle in their subtree, which is easily found by selecting the larger circle of the two children of a node. In the last pass, for each leaf, node-circles in the path from the leaf up to the root are modified - if needed - such that they fully overlap with the leaf-circle. (Passes two to four could be replaced by just combining pairs of circles bottom up, but this gives inferior results). Passes one to three take $O(N)$ steps, the last pass takes

$O(N \log N)$ steps, as the paths from leaf to root have $O(\log N)$ length. In general, the bounding circles calculated by this method will be 10 to 15% larger than the smallest enclosing circle of the set of particles in the subtree, but this method is significantly faster than the exact method and - as experiments have shown - does not significantly degrade the speed of force calculations carried out based on the tree.

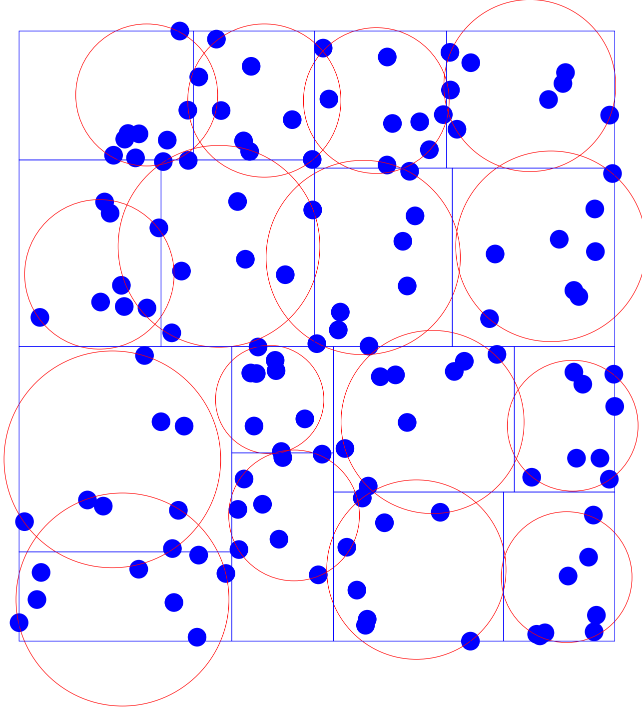

Fig. 1. k-d-tree like partioning of a set of particles into leaf rectangles and corresponding bounding circles

Figure 1 shows for a small example the partitioning of the plane into rectangular areas achieved by building the 2-d-tree from a set of particles. We see also the enclosing circles for the leaf nodes of the tree. Note that in general, these circels are smaller than the circumscribed circle of the corresponding leaf-rectangle, that had to be used for distance calculations if our circles were not available.

3.3 Tree Update

In the main loop of the force-directed spring embedder, particles are moved according to the total force acting on them. Due to the multi-level approach

taken, these movements are small because particle positions are initialized with reasonable values derived from the previous level in the hierarchy of coarsened graphs. Therefore, there is no need to create the tree from scratch for each iteration. (It *is* created from scratch for each level of the hierarchy). Instead, we keep the tree structure and just update the enclosing circles in leafs and interior nodes, provided the average movement of particles in the last iteration was small (less than 10% of of the average leaf radius). For each particle, we follow the path from its leaf to the root of the tree and check if it is still inside the enclosing circle of the node. If not, the enclosing circle is appropriately enlarged. Otherwise, we are done with this particle as nodes further up cannot be affected. Typically, this loop terminates already at the bottom level and the typical runtime is 10 times smaller than that for building the tree from scratch. Nevertheless, we may rebuild the graph every, say 10th, iteration in order to avoid too much inaccuracy of bounding circles, without affecting speed significantly.

This speedup strategy is especially valuable at the last of the multi-level steps where the full graph is processed and node movements are small.

3.4 Tree Storage

As the size of the tree is known from the very beginning and its structure does not change, it is convenient to store the tree (actually pointers to interior tree nodes and leafs) in an array, just as a binary heap is conventionally stored: for a node at position i we find its predecessor at position $i/2$ and its children at positions $2i$ and $2i+1$. Thus we save three pointers per node.

4 Calculation of Repulsive Forces

As invented by Greengard [9] and thoroughly discussed by Hachul [10] our calculation of repulsive forces is based on $p-$ *term multipole expansions* of the potential energy due to groups of charged particles represented by nodes of our tree.

4.1 Calculation of p-Term Multipole Expansions

In what follows, coordinates, forces, and coefficients are complex numbers. For coordinates, we can convert between two-dimensional vectors and complex numbers in the obvious way, for forces we have to take the conjugate first. Formulas are taken from [10] where also proofs have been given.

The p-term multipole expansion with parameter p

$$e(z) = a_0 \log(z - z_0) + \sum_{k=1}^{p} \frac{a_k}{(z-z_0)^k}$$

with coefficients

$$a_0 = m \quad \text{and} \quad a_k = -\sum_{i=1}^{m} \frac{(p_i - z_0)^k}{k}$$

approximates the potential energy at point z with $|z - z_0| > r$ due to m particles with unit charge located in the plane at points p_i within a circle with radius r around z_0.

From the p-term multipole expansion we can derive the approximate force $f(z)$ that acts on a particle with unit charge at position z outside the circle (z_0, r) as

$$f(z) = \frac{a_0}{(z-z_0)} - \sum_{k=1}^{p} \frac{k \cdot a_k}{(z-z_0)^{k+1}}$$

For the efficient calculation of p-term multipole expansions in the nodes we also need to know that a p-term multipole-expansion around center z_0 can be shifted to a new position z_1. The coefficients b_i for the shifted expansion are obtained by

$$b_0 = a_0 \quad \text{and} \quad b_l = \frac{-a_0(z-z_1)^l}{l} + \sum_{k=1}^{l} a_k (z_0 - z_1)^{l-k} \binom{l-1}{k-1}$$

Using these formulas, we first calculate the coefficients $a_0 \dots a_p$ of the p-term multipole expansion for each leaf of the tree. Then, traversing the tree bottom up, we shift the expansions of the children of a node to the node's center and add the two sets of coefficients. The whole process takes $O(p^2 N \log N)$ steps.

In [10] in addition to multipole expansions so called *local expansions* are defined and used for force calculations. To simplify the implementation we do not, however, use local expansions.

4.2 Using p-Term Multipole Expansions

The multipole expansions (i.e., their coefficients) are calculated once for each iteration of the force-directed placement, after the tree of particles has been built or updated. Force-calculation is then a quite simple step. We loop over all leafs L1 of the tree and proceed as follows:

- For the b particles in the leaf's bucket, we calculate the mutual repulsive forces directly; this takes $b(b-1)/2$ steps.
- Then we need to calculate the forces from the other particles in the tree. We use a stack of interior nodes or leafs to be processed which is initialized with the root node. While the stack is not empty we pop an interior node or leaf and compare it with the current leaf L1. If the popped element is sufficiently far away, we use its multipole expansion to find the forces acting on the particles of the current leaf. Otherwise, if it is an interior node, its two children are pushed on the stack for further processing. If it is a leaf L2, we loop over the particles in L1 and either, if L2 is sufficiently far away from a particle, calculate the forces acting on it due to the multipole expansion of L2, or else calculate the forces acting on the particle from those of L2 directly. In the first case we need p calculations per particle, in the latter b calculations, with p the parameter of the p-term-multipole expansion and b the number of particles in leaf L2.

The partial forces calculated as described are accumulated in the respective particles, so that we end up with the total repulsive force acting on each particle.

It remains to specify what "sufficiently far away" means. A particle is considered sufficiently far away from a node with radius R of its bounding circle if its distance from the node's center is $\geq 2R$. A leaf with radius r is considered sufficiently far away if the same condition holds for all points enclosed by the leaf's bounding circle, i.e., if for the distance d between the two centers we have $d > 2R + r$. We tried to parameterize these conditions as in [6], as weaker distance requirements would speed up calculations, but this leads to a rapid degradation of accuracy if the node's circular bounds have been calculated exactly. If, however, the faster approximation is used for the calculation of node circles (that leads to larger bounds), we can compensate for this by multiplying node radii by 0.9 without degrading accuracy of force calculations too much.

As we have seen, we use explicit distance calculations between leafs and/or nodes to steer the algorithm instead of relying on implicit distance guarantees that come with quadtrees but require evolved bookkeeping of possibly interacting nodes [10]. We gain in simplicity and - as we will see, in speed - but we loose the possibility to give complexity results for this step. Thus, the efficiency of the method is shown by experimental results.

5 Experimental Result

The proposed algorithms have been implemented in C++ using our own C++-class library of basic algorithms and data structures [13] and as part of a spring embedder project. As the runtime of a spring embedder depends on many factors, e.g., local cooling schemes, termination conditions, translation of forces into particle movements, we focus here on the runtime for building and updating the particle tree and for the calculation of repulsive forces. Hachul [10] has made thorough comparisons of his reduced bucket quadtree based implementation with a number of other approximative methods and has shown his approach to be superior in all cases. It seems therefore justified to compare with his results only. For this purpose, we use the same particle distributions and numbers. We also use a similar hardware- and software platform (Intel Pentium 4 running Linux), however with a different clock rate. To make results comparable, we scaled our CPU-times such that running times for a naive $O(N^2)$ force calculation become identical.

5.1 Particle Distributions

We use these particle distributions as defined by Hachul [10]:

Uniform. Particles are uniformly distributed within the square $[0,1] \times [0,1]$.

Non-uniform. 20% of the particles are uniformly distributed within the square $[0,1] \times [0,1]$. The remaining particles are to equal parts distributed within circles with their center at $(\frac{1}{2}, \frac{1}{2})$ and radius $\frac{1}{4}$, $\frac{1}{16}$, $\frac{1}{64}$, and $\frac{1}{256}$, respectively.

Quasi-converging. Here particles are distributed on the line connecting (0,0) and $P = (0, 10^{25})$. Particle i is placed at position $\frac{\frac{3}{4}P}{2^i - 1}$ for $i = 1, 2, \dots$ until x_i becomes $< 10^{-25}$. The remaining particles are uniformly distributed on the line connecting (0,0) and $P = (10^{-25}, 10^{-25})$.

5.2 Running Times for Tree Construction

For the distributions described and particle numbers between 8000 and 128000 the particle tree was built and compared with the faster tree construction method TC_b of [10]. The bucket size was fixed at 16 for all runs. As expected, the running times are independent of the distribution, with one exception: due to the correlation between x- and y-coordinates in the quasi-convergent distribution, the second pass of sorting in our method becomes very fast. The measured running times confirm the expected complexity of $O(N \log N)$ for all distributions. In general, our tree construction method is slower than that for the quadtree, with the exception of the quasi-convergent distribution, where the quadtree is significantly slower. However, in the spring-embedder application, in most cases the tree can be updated instead of building it from scratch, which is about 10 times faster.

To clarify, the running times in Table 1 show the time for building the tree once from scratch. This has to be done once per iteration in the implementation of [10], but is done only every 10th iteration in our approach; so the actual time spent on tree building is significatly smaller than the table seems to indicate.

Table 1. Running time for building the tree data structure (Note that the time needed for dynamically updating the tree after small movements of particles is by a factor of about 10 lower.)

Distribution	Number of particles	Hachul's	Our's
uniform	8000	0.01	0.03
	16000	0.03	0.08
	32000	0.06	0.17
	64000	0.12	0.38
	128000	0.30	0.83
non-uniform	8000	0.02	0.03
	16000	0.03	0.08
	32000	0.08	0.17
	64000	0.15	0.38
	128000	0.34	0.84
quasi-converging	8000	0.22	0.02
	16000	0.44	0.06
	32000	0.71	0.13
	64000	1.49	0.27
	128000	2.75	0.64

5.3 Running Times for Force-Calculations

Now we compare the running times for calculation of repulsive forces. In [10], measurements were made with different parameter settings giving low, medium, and high accuracy, the latter being defined by an approximation error below 10^{-4}. We restrict our comparisons to this high accuracy case, which is achieved - as in [10] - by setting the parameter p of the multipole expansion equal to 6, 7, and 8 for the uniform, non-uniform, and quasi-converging distributions, respectively. The bucket size of the tree was again fixed at 16. Note that the times given refer just to the force calculations, excluding tree building and updating.

Table 2. Running time for calculation of repulsive forces

Distribution	Number of particles	Hachul's	Our's	Exact
uniform	8000	0.29	0.13	7.99
	16000	0.50	0.28	34.92
	32000	1.29	0.66	142.16
	64000	2.26	1.37	568.64
	128000	5.54	3.14	2274.56
non-uniform	8000	0.32	0.27	7.80
	16000	0.61	0.57	35.14
	32000	1.40	1.18	142.16
	64000	2.58	2.55	568.64
	128000	5.88	5.54	2274.56
quasi-converging	8000	0.29	0.16	7.83
	16000	0.60	0.34	35.08
	32000	1.06	0.71	142.16
	64000	2.13	1.45	568.64
	128000	4.00	3.13	2274.56

We see that our running times for force calculations for the uniform and for the quasi converging distribution are significantly lower than in the quadtree based approach and slightly smaller for the non-uniform distribution.

6 Conclusions

We have introduced a new particle tree variant, that stores smallest enclosing circles in its nodes and can be built fully balanced in $O(N \log N)$ steps independent of the particle distribution. After small movements of particles the tree can efficiently be updated, without any changes to its structure. Using explicit distance calculations between leafs and nodes and avoiding the calculation of local expansions, we achieve an easy to implement multipole expansion based approximation method for calculation of repulsive forces that compares - concerning speed and implementation efford - favorably with earlier work.

Acknowledgements. We are indebted to Stephan Hachul for many fruitful discussions of his work and our own evolving implementation. Thanks goes also to referees for helpful comments and suggestions.

References

1. Eades, P.: A heuristic for graph drawing. Congressus Numerantium **42** (1984) 149–160
2. Kamada, T., Kawai, S.: An Algorithm for Drawing General Undirected Graphs. Information Processing Letters **31** (1989) 7–15
3. Fruchterman, T., Reingold, E.: Graph Drawing by Force-directed Placement. Software–Practice and Experience **21** (1991) 1129–1164
4. Davidson, R., Harel, D.: Drawing Graphs Nicely Using Simulated Annealing. ACM Transaction on Graphics **15** (1996) 301–331
5. Hachul, S., Jünger, M.: Drawing large graphs with a potential-field-based multilevel algorithm. In Pach, J., ed.: Graph Drawing. Volume 3383 of Lecture Notes in Computer Science., Springer (2004) 285–295
6. Barnes, J., Hut, P.: A hierarchical $\mathcal{O}(N \log N)$ force-calculation algorithm. Nature **324** (1986) 446–449
7. Quigley, A., Eades, P.: FADE: Graph Drawing, Clustering, and Visual Abstraction. In Marks, J., ed.: Graph Drawing 2000. Volume 1984 of Lecture Notes in Computer Science., Springer-Verlag (2001) 197–210
8. Tunkelang, D.: JIGGLE: Java Interactive Graph Layout Environment. In Whitesides, S.H., ed.: Graph Drawing 1998. Volume 1547 of Lecture Notes in Computer Science., Springer-Verlag (1998) 413–422
9. Greengard, L.: The Rapid Evaluation of Potential Fields in Particle Systems. ACM distinguished dissertations. The MIT Press, Cambridge, Massachusetts (1988)
10. Hachul, S.: A Potential-Field-Based Multilevel Algorithm for Drawing Large Graphs. PhD thesis, Institut für Informatik, Universität zu Köln, Germany (2005) URN: `urn:nbn:de:hbz:38-14097`, URL: `http://kups.ub.uni-koeln.de/volltexte/2005/1409`.
11. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18** (1975) 509–517
12. Welzl, E.: Smallest enclosing disks (balls and ellipses). New Results and New Trends in Computer Science **555** (1991) 359–370
13. Lauther, U.: The c++ class library turbo - a toolbox for discrete optimization. Software@Siemens (2000) 34–36