Journal of Graph Algorithms and Applications
http://jgaa.info/
vol. 7, no. 3, pp. 253–285 (2003)

A Multilevel Algorithm for Force-Directed Graph-Drawing

Chris Walshaw
School of Computing and Mathematical Sciences,
University of Greenwich,
Old Royal Naval College, Greenwich,
London, SE10 9LS, UK.
http://www.gre.ac.uk/∼c.walshaw
C.Walshaw@gre.ac.uk

Abstract

We describe a heuristic method for drawing graphs which uses a multilevel framework combined with a force-directed placement algorithm. The multilevel technique matches and coalesces pairs of adjacent vertices to define a new graph and is repeated recursively to create a hierarchy of increasingly coarse graphs, $G_0, G_1, \ldots, G_L$. The coarsest graph, G_L, is then given an initial layout and the layout is refined and extended to all the graphs starting with the coarsest and ending with the original. At each successive change of level, l, the initial layout for G_l is taken from its coarser and smaller child graph, G_{l+1}, and refined using force-directed placement. In this way the multilevel framework both accelerates and appears to give a more global quality to the drawing. The algorithm can compute both 2 & 3 dimensional layouts and we demonstrate it on examples ranging in size from 10 to 225,000 vertices. It is also very fast and can compute a 2D layout of a sparse graph in around 12 seconds for a 10,000 vertex graph to around 5-7 minutes for the largest graphs. This is an order of magnitude faster than recent implementations of force-directed placement algorithms.

Keywords: graph-drawing, multilevel optimisation, force-directed placement.

Communicated by Michael Kaufmann: submitted March 2002; revised August 2003.

1 Introduction

Graph-drawing algorithms form a basic enabling technology which can be used to help with the understanding of large sets of inter-related data. By presenting data in a visual form it can often be more easily digested by the user and both regular patterns and anomalies can be identified. However most data sets do not contain any explicit information on how they should be laid out for easy viewing, although normally such a layout will depend on the relationships between pieces of data. Thus if we model the data points with the vertices of a graph and the relationships with the edges we can use graph-based technology and, in particular, graph-drawing algorithms to infer a 'good' layout from an arbitrary data set based on the relationships.

There has been considerable research into graph-drawing in recent years and a comprehensive survey can be found in [2]. Many such algorithms are based on physical models and the vertices are placed so as to minimise the 'energy' in the physical system (see below, §2.3). Typically such algorithms are able to display structures and symmetries in the graph but their computational cost in terms of CPU time is very high.

1.1 Motivation

The motivation behind our approach to graph-drawing arises from our work in the field of graph partitioning and the multilevel paradigm, e.g. [20, 21]. In recent years it has been recognised that an effective way of enhancing partitioning algorithms is to use multilevel techniques and this strategy has been successfully developed to overcome the localised nature of the Kernighan-Lin and other partition optimisation algorithms, e.g. [12]. The multilevel process has also recently been successfully applied to the travelling salesman and graph colouring problems and appears to work (for combinatorial problems at least) by sampling and smoothing the objective function, [20], thus imparting a more global perspective to the optimisation.

This is an important consideration for graph-drawing; the localised positioning of a vertex relative to fixed neighbours is actually fairly easy and it is the global untangling of the graph which is more difficult or time consuming. We therefore aim to use the multilevel ideas to both enhance the layout and accelerate the graph-drawing process.

In this paper (and an earlier version, [19]) we apply multilevel ideas to force-directed placement (FDP) algorithms. In fact such ideas have been previously suggested in the graph-drawing literature and for example in 1991 Fruchterman & Reingold, [7], suggested the possible use of 'a multigrid technique that allows whole portions of the graph to be moved', whilst Davidson & Harel, [1], suggest a multilevel approach to 'expedite the SA [simulated annealing] process'. More recently Hadany & Harel, [9], and in particular Harel & Koren, [10], have actually used multilevel ideas (or as they refer to them, *multiscale*) and are able to robustly handle graphs of up to 15,000 vertices. However their algorithm uses the placement scheme of Kamada & Kawai, [13], which requires the graph theoretic distances (path lengths) between pairs of vertices, and hence the overall complexity of the method contains an $O(N^2)$ term. Gajer *et al.*, who subsequently developed a similar scheme, [8], managed to reduce this complexity by calculating these distances dynamically and also enhanced the scheme by computing the layout in higher dimensions. All three of these approaches ([8, 9, 10]) share many features with the algorithm outlined here (although derived independently) and confirm that the multilevel paradigm can be a powerful tool for force-directed placement irrespective of the specific FDP algorithm used.

A related but somewhat different idea is that of multilevel drawings, e.g. [3, 6]. Rather than using the multilevel process to create a good layout of the original graph, a multilevel graph is created, either by natural clustering which exists in the graph or by artificial means similar to those applied here. Each level is drawn on a plane at a different height and the entire structure can then be used to aid understanding of the graph at multiple abstraction levels, [5].

Finally, although not strictly related to the multilevel ideas described here, it is worth mentioning that Koren *et al.* have recently developed a number of other graph-drawing schemes which can work even faster than multilevel force-directed placement (although the layout quality is often somewhat inferior). In particular, these include the use of the algebraic multigrid techniques, [14], and (building on the ideas due to Gajer *et al.*, [8]) the development of higher-dimensional embeddings, [11].

2 A multilevel algorithm for graph-drawing

In this section we describe how we combine the multilevel optimisation ideas with our variant of a force-directed placement algorithm.

2.1 Notation and Definitions

Let $G = G(V, E)$ be an undirected graph of vertices V, with edges E and which we will assume is connected. For any vertex v let Γ_v be the neighbourhood of, or set of vertices adjacent to, v, i.e., $\Gamma_v = \{u \in V : (u, v) \in E\}$. We use the $|.|$ operator to denote the size of a set so that $|V|$ is the number of vertices in the graph and $|\Gamma_v|$ is the number of vertices adjacent to v (the *degree* of v). We also use $|.|$ to denote the weight of a vertex; since weighted vertices in the coarsened graphs represent sets of vertices from the original graph, the weight of a coarsened vertex is just equivalent to the number of original vertices in the set it represents. We then use $||.||$ to denote Euclidean distance in either 2D or 3D.

2.2 The multilevel framework

As stated above, the inspiration behind our graph-drawing scheme is the multilevel paradigm, e.g. [20]. The idea is to coalesce *clusters* of vertices to define a new graph and recursively iterate this procedure to create a hierarchy of increasingly coarse graphs, $G_0, G_1, \ldots$ and until the size of the coarsest graph falls below some threshold. The coarsest graph, G_L, is then given an initial layout and the layout is refined and extended to all the graphs starting with the coarsest and ending with the original. At each successive change of level, l, the initial layout for G_l is taken from its coarser and smaller child graph, G_{l+1}, and refined using force-directed placement. Thus the algorithm does not actually operate simultaneously on multiple levels of the graph (as, for example, a multigrid algorithm might) but instead refines the layout at each level and then extends the result to the next level down.

2.2.1 Graph coarsening

There are many ways to create a coarser graph $G_{l+1}(V_{l+1}, E_{l+1})$ from $G_l(V_l, E_l)$ and clustering algorithms are an active area of research within the field of graph-drawing amongst others, e.g. [3, 17]. Usually such clustering algorithms seek to retain the more important structural features of the graph in order that the visualisation of each level is meaningful in itself. However, here we are only interested in the drawing of the original graph and as such we seek a fast and efficient (i.e., not necessarily optimal) algorithm that judiciously reduces the size of the graph. Thus, if too many vertices are clustered together in one step it may depreciate the benefits of the multilevel paradigm and in particular inhibit the force-directed placement algorithm, as applied to G_l, from making use of the positioning obtained for G_{l+1}. Conversely, if each clustering only shrinks the graph by a small fraction, the multilevel scheme may be significantly slowed by having to compute the layout for a multitude of fairly similar coarse graphs. To suit these requirements we choose (as is typical for partitioning) a coarsening approach known as *matching* in which each vertex is matched with at most one neighbour, so that clusters are thus formed of at most two vertices and the number of vertices in the coarsened graph G_{l+1} is no less than half the number in G_l.

Computing a matching is equivalent to finding a maximal independent subset of graph edges which are then collapsed to create the coarser graph. The set is independent if no two edges in the set are incident on the same vertex (so no two edges in the set are adjacent), and maximal if no more edges can be added to the set without breaking the independence criterion. Having found such a set, each selected edge is collapsed and the vertices, $u_1, u_2 \in V_l$ say, at either end of it are merged to form a new vertex $v \in V_{l+1}$ with weight $|v| = |u_1| + |u_2|$.

The problem of computing a matching of the vertices is known as the maximum cardinality matching problem. Although there are optimal algorithms to solve this problem, they are of at least $O(|V|^{2.5})$, e.g. [15]. Unfortunately this is too slow for our purposes and, since it is not essential for the multilevel process to solve the problem optimally, we use a variant of the edge contraction heuristic proposed by Hendrickson & Leland, [12]. Their method of constructing a maximal independent subset of edges is to create a randomly ordered list of the vertices and visit them in turn, matching each unmatched vertex with an unmatched neighbouring vertex (or with itself if no unmatched neighbours exist). Matched vertices are removed from the list.

If there are several unmatched neighbours the choice of which to match with can be random, but in order to keep the weight of the vertices in the coarser graphs as uniform as possible, we choose to match with the neighbouring vertex with the smallest weight (note that even if the original graph G_0 is unweighted, G_l for $l = 1, 2, \ldots$ will be weighted). In the case of several such minimally weighted neighbours a random choice is made from amongst them. Other matching heuristics were tested (e.g. such as one that prefers to match across heavily weighted edges) but did not reveal any noticeable benefits and in the end the choice was based purely on empirical evidence (not presented here).

2.2.2 The initial layout

Having constructed the series of graphs until the number of vertices in the coarsest graph, G_L, is smaller than some threshold, we need to compute an initial layout for G_L. However, if the graph is coarsened down to 2 vertices (which because of the mechanisms of the coarsening will be connected by a single weighted edge) we can simply place these vertices at random with no loss of generality.

Note that contraction down to 2 vertices should always be possible provided the graph is connected (assumed, §2.1). To see this consider that every connected graph of $|V|$ vertices must have at least $|V| - 1$ edges and that the collapsing of an edge results in a connected graph. Thus, if $|V| > 2$ there must be at least one edge which can be collapsed to create a graph with $|V| - 1$ vertices and so on by induction.

2.2.3 Uncoarsening

At each level l the layout on graph $G_l(V_l, E_l)$ is refined and then extended to its parent $G_{l-1}(V_{l-1}, E_{l-1})$. This uncoarsening step is a trivial matter and matched pairs of vertices, $v_1, v_2 \in V_{l-1}$, are placed at the same position as the cluster, $v \in V_l$, which represents them.

2.3 The force-directed placement algorithm

We use a standard drawing algorithm to refine the layout on the graph, G_l, at each level l. There has been considerable research into graph-drawing paradigms, [2], and here we are interested in *straight-line* drawing schemes and, in particular, *spring-embedder* or *force-directed placement* algorithms. The original concept came from a paper by Eades, [4], and is based on the idea of replacing vertices by rings or hinges and edges by springs. The vertices are given initial positions, usually random, and the system is released so that the springs move the vertices to a minimal energy state (i.e., so that the springs are compressed or extended as little as possible).

Unfortunately the local spring forces are insufficient to globally untangle a graph and so such algorithms also employ global repulsive forces, calculated between every pair of vertices in the graph, and thus the system resembles an n-body problem. Such repulsive forces between non-adjacent vertices do not have an analogue in the spring system but are a crucial part of the spring-embedder algorithms to avoid minimal energy states in which the system is collapsed in on itself in some manner. As a simple example of this consider a chain of 3 vertices $\{u, v, w\}$ connected by two edges (u, v) and (v, w) and a spring model of this graph where both springs have a natural length k. Perhaps the most intuitive zero energy layout for this system would have u & w placed a distance $2k$ apart with v in the middle. However, with no global repulsive forces there is nothing to stop u & w from being placed in the ***same*** position and if this is a distance k away from v then once again the energy is zero. On a larger scale, repulsion is necessary to push whole regions, which are not immediately connected, away from each other.

The particular variant of force-directed placement that we use is based on an algorithm by Fruchterman & Reingold (FR), [7], itself a variation of Eades' original algorithm. From the point of view of the multilevel approach it is attractive as it is an incremental scheme which iterates to convergence and which can reuse a previously calculated initial layout. We have made a number of parameter modifications based on our experience with it and, in particular, because of the additional problems associated with drawing very large graphs. In principle however, it should be possible to use any iterative incremental algorithm for this part of the multilevel graph-drawing, although in practice different algorithms can be somewhat sensitive and require a certain amount of tuning.

Figure 1 shows the basic outline of our algorithm and is written in a similar fashion to the original FR algorithm, [7]. Thus Δ is shorthand notation for the difference vector between the positions of two vertices and Θ is short for the vector of displacements calculated for the current vertex v. There are two main differences (apart from the choice of parameters); the order of updating and the weighting of the repulsive forces (discussed in more detail below). One other fairly minor difference is that we do not impose any boundaries around the drawing (referred to as the frame in [7]); the layout can thus expand (or contract) as required by the forces within the system. The positions may be subsequently scaled to fit onto a computer screen or a hardcopy or indeed into any region required by the user, but this forms no part of the algorithm.

2.3.1 Updating

An important difference from the original FR algorithm is the order of updating of the vertex positions. The original algorithm used two vectors (of length $|V|$), one containing the position of the vertices and the second containing their displacement as calculated during the current iteration of the outer loop. The outer loop then contained three main inner loops, the first looping over the vertices to calculate displacement caused by (global) repulsive forces and the

```
{ initialisation }
function $f_r(x, w)$ := begin return $-Cwk^2/x$ end
function $f_a(x)$ := begin return $x^2/k$ end
$t := t_0$;
$Posn := NewPosn$;

while ($converged \neq 1$) begin
    $converged := 1$;

    for $v \in V$ begin
        $OldPosn[v] = NewPosn[v]$
    end

    for $v \in V$ begin
        { initialise $\Theta$, the vector of displacements of $v$ }
        $\Theta := 0$;

        { calculate (global) repulsive forces }
        for $u \in V$, $u \neq v$ begin
            $\Delta := Posn[u] - Posn[v]$;
            $\Theta := \Theta + (\Delta/||\Delta||) \cdot f_r(||\Delta||, |u|)$;
        end

        { calculate (local) attractive/spring forces }
        for $u \in \Gamma_v$ begin
            $\Delta := Posn[u] - Posn[v]$;
            $\Theta := \Theta + (\Delta/||\Delta||) \cdot f_a(||\Delta||)$;
        end

        { reposition $v$ }
        $NewPosn[v] = NewPosn[v] + (\Theta/||\Theta||) \cdot \min(t, ||\Theta||)$;
        $\Delta := NewPosn[v] - OldPosn[v]$;
        if ($||\Delta|| > k \cdot tol$) $converged := 0$;
    end

    { reduce the temperature to reduce the maximum movement }
    $t := cool(t)$;
end
```

Figure 1: Force-directed placement algorithm

second looping over edges and calculating the displacement (on the vertices at either end of the edge) due to the local attractive forces. The final inner loop over the vertices updated the positions.

In our version however, we only calculate displacements for one vertex at a time, updating each at the end of the inner loop. At first it might seem as if this is less efficient, since the attractive forces are calculated twice for each edge. However, *Posn* is a pointer which points to *NewPosn*, the newly calculated position of each vertex which may have already been updated during the ***current*** iteration of the outer loop. In our experience this dramatically improves the performance of the algorithm (see §3.2). It is also very easy to recover the behaviour of the original FR algorithm (for comparison) by setting the pointer *Posn* := *OldPosn* in the initialisation section.

2.3.2 Vertex weighting.

We use a weighted version of the original FR repulsive function, computed by multiplying the repulsive force by the weight, $|u|$, of the vertex, u, which generates it, to give $f_r(x, |u|) = -C \cdot |u| \cdot k^2/x$. Although we are typically (but not exclusively) interested in drawing unweighted graphs, any of the coarsened graphs will have weights attached to both vertices & edges and in particular the vertex weight of a coarsened vertex u will represent the sum of weights of vertices from the original graph contained in the cluster. If we then consider the repulsive forces in the original graph, all of the vertices in the cluster u would act on any vertex from the cluster v so it makes sense to multiply the repulsive force of u on v by $|u|$. This was also confirmed by experimentation and made a considerable improvement as compared with neglecting this factor. For unweighted graphs, and in particular the force-directed algorithm used in its standard single-level format then $|u| = 1$ for all $u \in V$ and this function reverts back to the original FR version from [7].

Finally the constant C was determined by experimentation as suggested by Fruchterman & Reingold. We found that the smaller the value of C, the better the algorithms (both multilevel and the original single-level version) seemed to work, but the longer they took to run. This is presumably because, with the grid-variant in use (see below, §2.4), the smaller the value of C, the smaller the effect of the repulsive forces and hence the more vertices are used to calculate them. Thus the quality improves but the runtime increases. After extensive testing we settled on $C = 0.2$, although $C = 0.5$ & $C = 0.1$ could equally be used to give similar results.

2.3.3 Edge weighting.

Note that there is no simple equivalent edge weight analogue for the local attractive forces. To see this consider the three graphs shown in Figure 2(a)-(c) and suppose that in each case the ringed vertices are matched and merged to give the graph shown in Figure 2(d). The weight of the edge in Figure 2(d) would then be 3 if derived from Figure 2(a), 2 if derived from 2(b) and 1 if

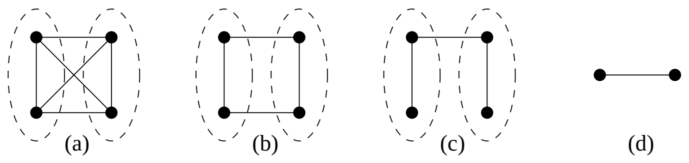


Figure 2: Examples of coarsening

from 2(c). Now consider the attractive force on each vertex of the cluster v by looking at the attractive forces in the three original graphs and assuming that the matched vertices are placed at the same position. In the case of the graph in Figure 2(a) there are 2 attractive forces from each vertex in the cluster w on each vertex in the cluster v (this corresponds to the edge weight 3). Meanwhile, for Figure 2(b) there is 1 attractive force per vertex (corresponding to edge weight 2). However for the graph in Figure 2(c) it is not even clear what the attractive force from cluster w on cluster v should be, although arguably it should be less than 1 in some averaged sense (and this corresponds to edge weight 1). Hence there is no linear relation between edge weight and attractive forces and indeed for more complex cases (i.e., after multiple coarsenings) the relationship becomes even harder to evaluate.

The simplest way of dealing with this problem, and the one that we use for all the experiments in this paper, is just to ignore edge weights. However, we have tested two alternative schemes (using the same testing regime as that described in Section 3). The first scheme we tried was to multiply each attractive force by the weight of the edge along which it acts. In fact this produced very similar results to ignoring the edge weights altogether, except that the drawings were somewhat less extended and took around 10-20% longer to compute (essentially both of these effects arise because the attractive forces are stronger relative to the repulsive ones and similar results can be seen simply by reducing the size of the parameter C).

The second alternative was to average the attractive forces by again multiplying each force by the corresponding edge weight but also dividing by the weight of the cluster on which it acts (giving multipliers of $\frac{3}{2}$, 1 & $\frac{1}{2}$ respectively for the graphs in Figures 2(a), (b) & (c)). In fact this produced worse layouts than ignoring the edge weights, especially when using a fast cooling schedule (see §2.3.5), although for slow cooling schedules it made little difference. In the end, however, we decided to ignore edge weights.

2.3.4 Natural spring length, k

A crucial part of the algorithm is the choice of the natural spring length, k, (the length at which a spring or edge is neither extended nor compressed). At the start of the execution of the placement algorithm for graph G_l the vertices will all be in positions determined by the layout calculated for graph G_{l+1} (except for G_L, the coarsest graph). We must therefore somehow set the spring length relative to this existing layout in order not to destroy it. If, for example, we set k too large, then the entire graph will have to expand from its current layout and potentially ruin any advantage gained from having calculated an initial layout via the multilevel process.

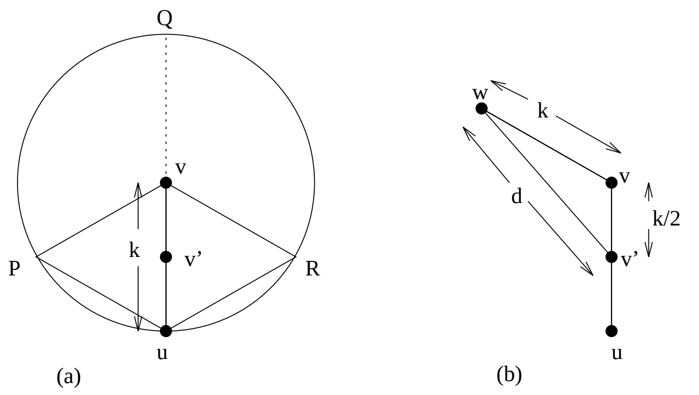


Figure 3: Calculation of natural spring length

In fact we derive the new value for k by considering what happens when we coarsen a graph, G_l, with well placed vertices (i.e., all vertices are approximately at a distance k from each other). Consider Figure 3(a) and suppose that v and u (at distance k from each other) are going to be clustered to form a vertex v' at the mid-point between them. Any vertex w adjacent to v should, if ideally spaced, lie somewhere on the arc PQR of the circle of radius k centred on v (it should not be on the arc PuR as that would place it too close to u). The distance between w and v' will then be $3k/2$ if w lies at Q or $\sqrt{3}k/2$ if w lies at P or R. If we take an average position for w midway along the arc PQ then from Figure 3(b) and the cosine rule, the length d is given by

$$d^2 = k^2 + \left(\frac{k}{2}\right)^2 - 2 \cdot k \cdot \frac{k}{2} \cos(2\pi/3) = k^2 + \frac{1}{4}k^2 + \frac{1}{2}k^2 = \frac{7}{4}k^2$$

If we take d as an estimate for the new natural spring length k' then $k' = \sqrt{7/4} \cdot k$.

Reversing this process, given a graph G_{l+1} with natural spring length k_{l+1}, we can estimate the natural spring length for the parent graph G_l at the start of the placement algorithm to be

$$k_l = \sqrt{\frac{4}{7}} \cdot k_{l+1}.$$

Remarkably this simple formula works very robustly over all the examples that we have tested, certainly better than other functions we have tried (for example based on average edge length of the initial layout). Very occasionally on one or two of the examples the value for k that it gives is too small for the existing layout and the graph placement expands rapidly for the first few iterations. However, this usually occurs on one of the coarser graphs and the multilevel procedure is still able to find a good layout. Nonetheless we feel that the choice of this parameter could do with further investigation.

For the initial coarsest graph, G_L, we simply set

$$k_L = \frac{1}{|E_L|} \sum_{(u,v) \in E_L} ||(u,v)||,$$

the average edge length. Typically we coarsen down to 2 vertices and 1 edge and so k is set to the length of that edge.

2.3.5 Convergence

We retain the 'cooling schedule' used in the original FR algorithm. Notice from Figure 1 that when the positions are updated, the maximum movement is limited by the value t (or temperature) and that t is reduced at the end of each iteration of the outer loop. This idea, drawn from a graph-drawing algorithm due to Davidson & Harel, [1] and based on simulated annealing, allows large movements (high temperature t) at the beginning of the iterations but progressively reduces the maximum movement as the algorithm proceeds (and the temperature falls). Fruchterman & Reingold do not give the exact cooling schedule that they use, although they do recommend a two phase scheme, first cooling rapidly and steadily (possibly linearly) and the second phase at a constant low temperature. Here for simplicity we use the scheme $t_i = \lambda t_{i-i}$, or in pseudo-code

```
**function** $cool(t)$ := **begin return** $\lambda t$ **end**
```

which operates similarly (i.e., initial rapid decay and a slow tail-off) but only involves one parameter, λ. After experimentation we then set $t_0 = k_l$ at each level l and the algorithm is then deemed to have converged when the movement of every vertex is less than some tolerance, *tol*, times k_l. Again after extensive experimentation we set $tol = 0.01$. This also allows us to avoid explicitly setting any maximum number of iterations since eventually the temperature will drop below $tol{\cdot}k_l$ and so there is an implicit limit.

By varying the cooling rate, λ, and measuring performance against runtime for a range of values of λ, we are able to compare different algorithms in a more meaningful way (see §3.2). However in the examples following we then recommend a value of $\lambda = 0.9$ and this means that all movement ceases at iteration i where $0.9^i < 0.01$ or in other words after 44 iterations. This is close to the 50 iterations recommended in the original FR algorithm, [7].

2.3.6 Coincident vertices

The algorithm needs minor exception handling if two vertices are found to be in exactly the same position. This can occasionally occur during the execution of the algorithm but it also always happens when the code commences on a graph G_l, having calculated the layout on G_{l+1}, since we initially place the vertices in a cluster at the same position as the cluster. In these cases the vertices are simply treated as if they were a small distance apart (the actual direction generated randomly with the distance no more than $0.001 \cdot k$) and the forces calculated accordingly. This allows us to extend the layout of one graph to its parent without any additional sophisticated mechanism.

2.4 Reducing the complexity

It is fairly clear from the description of the algorithm that the placement complexity for each iteration on graph $G_l(V_l, E_l)$ is $O(|V_l|^2 + |E_l|)$. For the types of sparse graphs in which we are interested, the $|V_l|^2$ heavily dominates this expression and we therefore use the FR grid variant for reducing the run-times, [7]. Their motivation was that over long distances the repulsive forces are sufficiently small to be neglected. If we set R to be the maximum distance over which repulsive forces will act we can then modify the algorithm by changing the global force calculation to:

```
function $f_r(x, w)$ :=
begin
        if $(x \leq R)$ return $-Cwk^2/x$;
        else return $0.0$;
end
```

In itself this modification will do little or nothing to speed up the calculation as the complexity is still $O(|V_l|^2)$. However Fruchterman & Reingold, [7], showed that if the domain is divided up into regular square cells (or cube shaped cells in 3D) of size R^2 (or R^3 in 3D) then each vertex will only be affected by repulsive forces from vertices in its own and adjacent cells (including those diagonally adjacent). To implement this efficiently we simply visit every vertex at the start of each outer loop and add each to a linked list of vertices for the cell to which it belongs. Repulsive forces can then be calculated for each vertex by using the linked lists of their own and adjacent cells. In practice this seems to work very well although we note that the number of grid cells can greatly exceed the number of vertices, particularly in 3D. However the implied memory limitations are not difficult to deal with by storing only the non-empty cells in a tree structure rather than storing all of them in an array.

We also note that, since we update vertex positions continuously throughout the outer loop, vertices are quite likely to move from one cell to another and thus not appear in the appropriate linked list. However we ignore the possible inaccuracies and do not transfer them during the course of an iteration and in practice it does not seem to matter.

Finally we must decide what value to give to R. In the original FR algorithm the value $R = 2k$ was used, but for the larger graphs in which we are interested, this did not prove sufficient to 'untangle' them in a global sense. Unfortunately the larger the value given to R the longer the algorithm takes to run and so although assigning $R = 20k$ gave better results, it did so with a huge time penalty. Fortunately, however, the power of the multilevel paradigm comes to our aid once again and we can make R a function of the level l. Thus for the initial coarse graphs we can set R_l to be relatively large and achieve some impressive untangling without too much cost (since $|V_l|$ is very small for these graphs). Meanwhile, for the final large graphs, when most of the global untangling has already been achieved we can make R_l relatively small without penalising the placement. In fact, provided this parameter is not too small it should be very robust (since it just determines a cut off point for tiny repulsive forces) and because the first such schedule that we tried worked very well, we have not experimented further.

The value that we use, therefore, is $R_l = 2(l + 1)k_l$ for each graph G_l. In this expression l is just the graph number where G_0 is the original graph and G_l the graph after l coarsenings. Conveniently this also replicates the choice of $R = 2k$ for G_0 in the original single-level FR algorithm.

2.5 Complexity analysis

It is not easy to derive complexity results for the algorithm but we can state some bounds. Firstly the number of graph levels, L, is dependent on the rate of coarsening. At best the number of vertices will be reduced by a factor of 2 at every level (if the code succeeds in matching every vertex with another one) and in the worst case, the code may only succeed in matching 1 vertex at every level (e.g. if the graph is a star graph, a 'hub' vertex connected to every other vertex each of which is only connected to the hub). Thus we have $\log_2 |V| \leq L < |V|$. This probably indicates that the algorithm is not well suited to graphs with a small diameter relative to their size (such as star graphs) and in fact for the examples given in Section 3 the coarsening rate is close to 2.

The matching & coarsening parts of the algorithm are $O(|V_l|+|E_l|)$ for each level l but in fact the total runtime is heavily dominated by the FDP algorithm. Using the above simplification (§2.4) of neglecting long range repulsive forces we can see that each iteration of the FDP algorithm is bounded below by $O(|V_l| + |E_l|)$ although with a large coefficient. In fact if the graph is dense, or in the worst case a complete graph, it may be that this is still $O(|V_l|^2+|E_l|)$, dependent on the relative balance of attractive & repulsive forces. However, we suspect that no FDP algorithm is appropriate for dense graphs because the minimal energy state corresponds to a tightly packed 'hair-ball' and so no structure is discernible in the drawing.

In summary the total complexity at each level is close to $O(|V_l| + |E_l|)$ for sparse graphs and the runtime is heavily dominated by the FDP iterations.

Finally consider the FDP algorithm used, without coarsening, on a given sparse graph of size N (i.e., standard single-level placement) and compare it with multilevel placement (MLFDP) used on the same graph. Let T_p be the time for the FDP algorithm to run on the graph and for MLFDP let T_c be the time to coarsen and contract it. If we suppose that the coarsening rate is close to 2 (which is true for most of the examples below) then for MLFDP this gives us a series of problems of size $N, N/2, \ldots, N/N$ whilst the (almost) linear complexity for the placement scheme at each level gives the total runtime for MLFDP as $T_c + T_p/N + \ldots + T_p/2 + T_p$. In all the examples we have tested $T_c \ll T_p$ and so we can neglect it giving a total runtime of approximately $T_p/N + \ldots + T_p/2 + T_p \approx 2T_p$. In other words MLFDP should take ***only*** twice as long as FDP to run (and yet in the examples below achieves far better results). In fact the final level of the MLFDP algorithm is likely to already have a very good initial layout which means that it should run even faster than FDP although this is neutralised somewhat by the fact that the coarsening rate is normally somewhat less than 2. Nonetheless this factor of 2 is a good 'rule of thumb' and note that if the chosen FDP algorithm were $O(N^2)$ or even $O(N^3)$ then a similar analysis suggests that the MLFDP runtime would be substantially ***less*** than twice that of FDP.

3 Experimental Results

We have implemented the algorithms described here within the framework of JOSTLE, a mesh partitioning software tool developed at the University of Greenwich[1]. We illustrate and test these schemes in a variety of ways and on a large number of problem instances including a suite of small random planar graphs together with some much bigger graphs from genuine applications. Firstly in §3.1 we demonstrate the multilevel scheme with an extended example of the technique in action. Next in §3.2 we present the results from extensive tests which show algorithmic performance against runtime and compare the behaviour of single-level and multilevel versions. In §3.3 we then present a test of runtime complexity and finally in §3.4 highlight the multilevel algorithm with some detailed individual layouts.

The experiments were all carried out using a 1 GHz Pentium III with 256 Mbytes of memory running Linux. (Although this is three times faster than the processor used for our original testing in [19], differences in floating point performance mean that it only runs about twice as fast on this sort of application. Other differences in runtime result from changes in the algorithms that we have made since the previous paper.)

3.1 An extended example

In this section we demonstrate in more detail how the multilevel scheme works. Figure 4 shows the original layout of a small mesh-based graph, 516 (with 516 vertices), drawn from a computational mechanics problem, and (lightly shaded) the underlying triangular mesh. Typically in such graphs the vertices can either represent mesh nodes (the nodal graph), mesh elements (the dual graph),

[1]freely available for academic and research purposes under a licensing agreement from `http://www.gre.ac.uk/jostle`

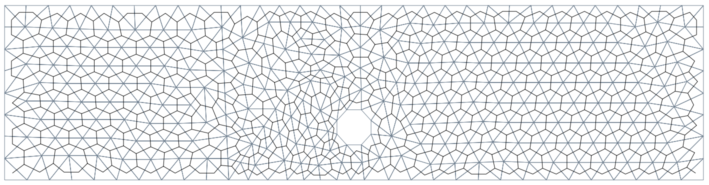


Figure 4: The original layout of 516 also showing the underlying triangular mesh elements

a combination of both (the full or combined graph) or some other special purpose representation. In this case the graph is a dual graph where each vertex represents a triangular element.

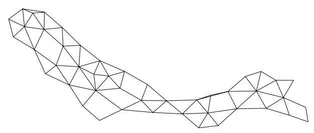
Table 1: The sizes of the coarsened graphs of 516

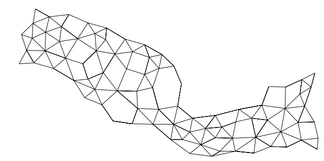
l	0	1	2	3	4	5	6	7	8	9
$\|V_l\|$	516	288	156	86	46	24	13	7	4	2
$\|E_l\|$	729	501	319	190	97	48	23	9	3	1

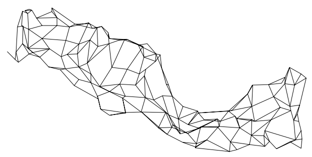
The MLFDP algorithm was applied to this problem (ignoring the existing layout) and Table 1 lists the sizes of the graphs, $G_1(V_1, E_1)$ to $G_9(V_9, E_9)$, constructed by the coarsening. Notice that $|V_l| \geq |V_{l-1}|/2$ since no more than two vertices are clustered together and so the graph cannot shrink by more than a factor of two. The initial layout is computed by placing the two vertices of G_9 at random and setting the natural spring length, k, to be the distance between them. Starting from $G_l = G_8$ the layout is extended from G_{l+1}, by simply placing vertices at the same position as the cluster representing them in the coarser graph, and then refined.

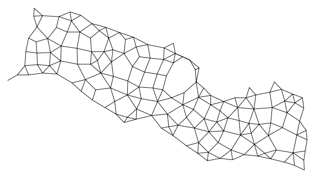
Figure 5(a) shows the final layout on G_4 and it can been clearly seen that, although over 10 times smaller than the original, the layout is already beginning to take shape. Figures 5(b)-(d) meanwhile illustrate the placement algorithm on G_2. Figure 5(b) shows the initial layout as extended from G_3 and with many of the vertices coincident whilst Figure 5(c) then shows the layout after the first iteration and where the coincident vertices have been pushed apart. Figure 5(d) finally shows the layout after the placement algorithm has converged for G_2. Notice an important feature of the multilevel process (common with the partitioning counterpart) that the final layout (partition), does not differ greatly from the initial one and hence the placement scheme at each level need not be very powerful in a global sense, since the multilevel framework seems to impart this property. Figure 5(e) shows the final layout on the original graph, G_0. The small kink arises from the hole in the graph which distorts the layout slightly, but in general the final drawing is excellent. Finally note that the MLFDP algorithm took less than half a second to compute this layout (this is the time for the entire algorithm including reading the problem, coarsening and placement at each level).

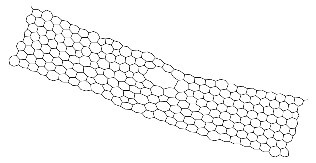
For comparison, Figure 5(f) shows the placement algorithm used on a random initial layout of the same graph (in other words as the standard single-level placement algorithm, FDP). Possibly the algorithm is not well tuned for this problem, but what can be seen is that although the micro structure of the graph has been reconstructed reasonably well (at least this can be seen by examining the layout in more detail than Figure 5(f) allows), the single-level placement has not been able to 'untangle' the graph in a global sense. In fact, by adjusting the cooling schedule so that the algorithm runs for at least 2,150 iterations, the single-level scheme can achieve a similar layout to that shown in Figure 5(e); however this takes 9.76 seconds to run, about twenty times longer than the 0.48 seconds required by the multilevel scheme. We believe that this at least hints at the power of the multilevel framework.


(a) MLFDP, G_4, final iteration


(b) MLFDP, G_2, iteration 0


(c) MLFDP, G_2, iteration 1


(d) MLFDP, G_2, final iteration


(e) MLFDP, G_0, final iteration

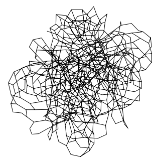

(f) FDP, final iteration

Figure 5: The multilevel algorithm illustrated for the graph 516

3.2 Comparison of single-level and multilevel algorithms

Although anecdotal evidence (above and in [19]) suggests that the multilevel framework can significantly enhance force-directed placement, we test this conclusion more thoroughly by comparing algorithmic performance on two test suites. The first suite[2] consists of 200 randomly generated planar graphs originally constructed to benchmark the GDT[3] software. There are 20 graphs each of size $|V| = 10, 20, \ldots, 100$ and we have subdivided the suite into two subclasses: 100 tiny graphs with $10 \leq |V| \leq 50$ and 100 small graphs with $60 \leq |V| \leq 100$.

Table 2: The test suite of mesh-based graphs

	size		degree			MLFDP placement ($\lambda = 0.9$)	
graph	$\|V\|$	$\|E\|$	max	min	avg	crossings	time (secs.)
140	140	199	3	2	2.84	0	0.12
grid1	252	476	4	2	3.78	121	0.29
399	399	573	3	1	2.87	0	0.38
rj20	400	760	4	2	3.80	0	0.42
516	516	729	3	1	2.83	0	0.48
771	771	1133	3	1	2.94	0	0.86
788	788	1123	3	2	2.85	90	0.78
mesh1024	1024	1504	3	2	2.94	0	1.18
dime01	1095	1570	3	2	2.87	34	1.03
sierpinski06	1095	2187	4	2	3.99	0	0.81
grid2	3296	6432	5	2	3.90	0	3.82
3elt	4720	13722	9	3	5.81	3391	7.57
uk	4824	6837	3	1	2.83	116	5.02
4970	4970	7400	3	2	2.98	0	6.86
dime06	5343	7836	3	2	2.93	295	5.53
ukerbe1	5981	7852	8	2	2.63	374	9.48
whitaker3	9800	28989	8	3	5.92	0	11.48
sierpinski08	9843	19683	4	2	4.00	158	7.71
t10k	10027	14806	3	2	2.95	0	10.65
crack	10240	30380	9	3	5.93	0	13.31

The second suite comprises 20 much larger planar graphs, listed in Table 2, and mostly drawn from genuine examples of computational mechanics meshes. The Table gives their sizes ($|V|$ & $|E|$) and the maximum, minimum & average degree of the vertices. It also shows the number of edge crossings and runtime required for a layout produced by the multilevel algorithm (using the cooling rate, $\lambda = 0.9$, suggested below, §3.3). Once again we have split the suite into two subclasses: 10 small mesh-based graphs of between 100 and around 1,000 vertices and 10 medium sized with between 1,000 to around 10,000 vertices. Note that although these graphs are all planar, some of them (in particular grid1, dime01, 3elt & dime06) are exceptionally difficult to draw with a planar layout because of extreme variations in mesh density. Meanwhile sierpinski06 & sierpinski08, despite their regular local structure, contain large holes which add to the drawing complexity. Both of these difficulties are explained further and illustrated in §3.4.

We use the test suites to compare three different algorithms: FDP$_0$, the original FR algorithm (or as close as we can get to it); FDP, our version of that algorithm; and MLFDP, the multilevel version of FDP. We have also tested

[2]available from `http://www.dia.uniroma3.it/~gdt/testsuite/GDT-testsuite-BUP.tgz`
[3]graph-drawing toolkit, see `http://www.dia.uniroma3.it/~gdt/`

MLFDP$_0$, the multilevel version of FDP$_0$, but since it always seems to perform worse than MLFDP we do not present any results for it.

The variants are differentiated by parameter settings defined as follows (in approximate order of importance):

FDP$_0$	FDP	MLFDP	
$Posn := OldPosn$	$Posn := NewPosn$	$Posn := NewPosn$	
$t_0 := 0.1\sqrt[d]{A}$	$t_0 := k$	$t_0 := k_l$	
$k := \sqrt[d]{A/\lvert V\rvert}$	$k := \frac{1}{\lvert E\rvert}\sum_{e \in E} \lvert\lvert e\rvert\rvert$	$k_L := \frac{1}{\lvert E\rvert}\sum_{e \in E} \lvert\lvert e\rvert\rvert$	
		$k_l := \sqrt{\frac{4}{7}} \cdot k_{l+1}$	$l = 0, \ldots, L-1$

Thus for FDP$_0$ (the original FR algorithm) the updating is based on vertex positions at the start of the loop rather than their current positions (see §2.3.1). The initial temperature is then given by $t_0 = 0.1\sqrt[d]{A}$ where A is the area of the initial layout and d $(= 2, 3)$ the dimension of the layout we wish to compute. Fruchterman & Reingold actually suggest t_0 as one tenth of the width of the drawing but since we generate the initial positions by random placement in the region $[0, 1]^d$, for 2D drawings this amounts to the same thing. Similarly the original scheme uses $k = \sqrt{A/|V|}$ but for 3D drawings this should naturally be $k = \sqrt[3]{A/|V|}$ (where A is then the volume of the region) to be dimensionally correct.

We compare the three algorithms by looking at how close each can come to some notionally 'optimal' layout in a given time. Although it is impossible to define an optimal drawing of any graph (because this is very much a subjective choice), nonetheless studies with real users have indicated that 'reducing the number of edge crossing is by far the most important aesthetic', [16], and we use this measure. Furthermore in order to compare different drawing algorithms it is necessary to bear in mind that for optimisation schemes such as these, typically the longer an algorithm is allowed for refinement, the better the layout it is likely to achieve. It is therefore insufficient to choose fixed parameter settings and compare results since the runtimes of the different algorithms are likely to be very different. We thus compare the algorithms over a range of different runtimes and fortunately, as described in §2.3.5, the cooling schedule, and in particular the cooling rate, λ, allows us an easy method to do this.

To assess a given algorithm then, we measure the runtime and solution quality (number of edge crossings) for a chosen group of problem instances and for a variety of values of λ. For problem instance p, at cooling rate λ, this gives a pair, $Q_{\lambda,p}$, the solution quality found, and $T_{\lambda,p}$, the runtime. We then normalise the runtime values and average over all problem instances to give a single data point of averaged solution quality, Q_λ, and runtime, T_λ, for a given cooling rate λ. By using several cooling rates, λ, we can then plot Q_λ against T_λ to give an indication of algorithmic performance over those instances.

Typically one might think of normalising solution quality by dividing the results by the quality of the optimal (or best known) solution, e.g. as in [20]. However all of the test graphs in this section are known to be planar and so an optimal layout (at least in terms of the performance measure) contains no edge crossings (i.e., quality 0). An alternative normalisation could be the number of edges in each graph, $|E|$, or even the number of edges squared since presumably $|E|^2$ is the maximum possible number of edge crossings. However we have not used either of these and, since the graphs in each subclass do not exhibit too much variation in size, we do not normalise the solution quality. The time normalisation is more simple and is calculated by $T_{\lambda,p}/T_p^{\mathrm{A}}$ where T_p^{A} is the runtime on an instance p for some well known reference algorithm, A. In this case we use A = FDP$_0$ with $\lambda = 0.9$.

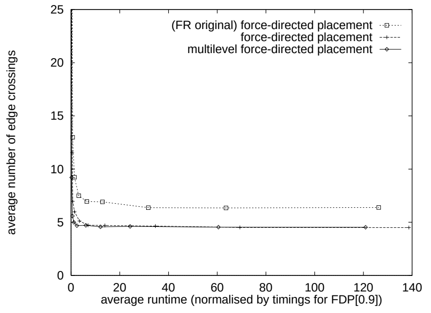
To summarise then, for a set of problem instances P, we plot averaged solution quality Q_λ against averaged normalised runtime T_λ for a variety of cooling rates, λ, and where:

$$Q_\lambda = \frac{1}{|P|}\sum_{p\in P} Q_{\lambda,p} \quad , \quad T_\lambda = \frac{1}{|P|}\sum_{p\in P} \frac{T_{\lambda,p}}{T_p^{\mathrm{A}}}.$$

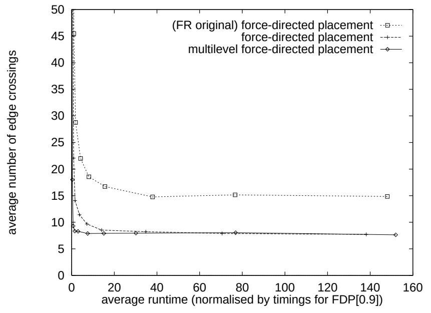
The particular cooling rates that we used for the tests shown here were

$$\lambda = 0.5, 0.8, 0.9, 0.95, 0.98, 0.99, 0.995, 0.998, 0.999$$

for MLFDP and additionally (because of the factor of two runtime overhead for MLFDP, §2.5) $\lambda = 0.9995$ for FDP$_0$ and FDP. In each case the runtime measurement includes reading in the problem, output of the solution and any initialisation required including an initial solution construction algorithm for the single-level local search schemes. It does not, however, include the time to count the edge crossings which forms no part of the algorithm and is only used here as a post-processed performance measure.



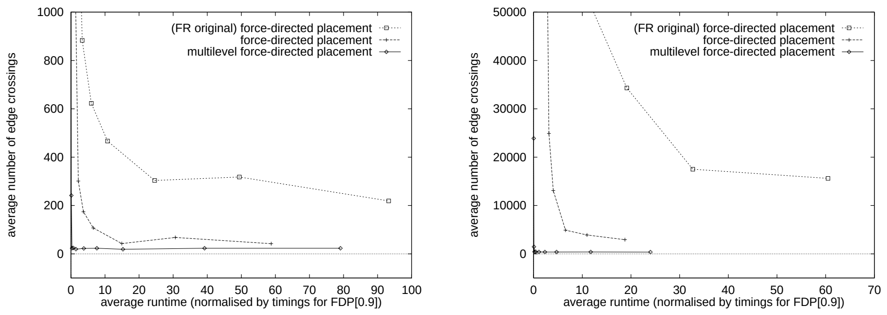
(a) tiny graphs ($10 \leq |V| \leq 50$)



(b) small graphs ($60 \leq |V| \leq 100$)

Figure 6: Plots of algorithmic behaviour on the random graphs

Figures 6 & 7 show the results on the two test suites and illustrate graphically the behaviour of the three algorithms. First of all we can see that the solution quality for FDP, our version of the FR algorithm, is far better than that for the original, FDP$_0$. The parameter settings certainly contribute to this improvement and hence this is a slightly unfair test for the original FR algorithm; firstly because Fruchterman & Reingold did not give complete parameter



(a) small graphs ($100 \leq |V| \leq 1,100$) (b) medium graphs ($1,000 \leq |V| \leq 11,000$)

Figure 7: Plots of algorithmic behaviour on the mesh-based graphs

settings and secondly because, even for those that they did give, our parameter settings were partially tuned on this set of problem instances and hence are likely to be more appropriate. On the other hand extensive testing of different parameter combinations revealed that by far the most significant contribution to the difference between FDP and FDP_0 was the fact that the positions are updated continuously rather than at the end of every outer iteration (see §2.3.1 for more details).

Next, comparing the single-level FDP algorithm with the multilevel version, we see that MLFDP can also significantly enhance the scheme and that the benefits increase with graph size. This is perhaps not a surprise since there is a much greater potential for global tangling in large graphs (rather than those with less than 50 vertices) and hence the multilevel scheme is more likely to be of assistance. Furthermore the larger the graph, the more coarsening, and hence the more refinement at different levels, takes place.

Looking at the curves in more detail in fact we see that for the random graphs, Figure 6, the MLFDP & FDP algorithms appear to have approximately the same asymptotic limit in solution quality. However the MLFDP curves bottom out more quickly.

The mesh-based graphs demonstrate this difference even more graphically in Figure 7 (where note that we have offset the x-axis away from 0 to avoid confusion with the MLFDP curve). MLFDP reaches its asymptotic limit almost immediately (for $\lambda \approx 0.9$) and it is not clear whether FDP can ever reach the same limit even after excessive runtimes (e.g. the 9,000 or so iterations represented by the final point on the FDP curves). Furthermore, even though the asymptotic limits for MLFDP & FDP do not seem that different, a look at individual layouts reveals that the single-level algorithm has not really untangled the graph properly. Of course the mesh-based test graphs are well structured locally and so we do not claim that the results would necessarily carry over to other graph types. On the other hand with such structure one might imagine that this sort of graph should be the ***easiest*** for a single-level force-directed approach to handle and that FDP$_0$ & FDP should be at their best for this suite.

Figure 7(a) demonstrates one further point; since the schemes are not directly trying to minimise the number of edge crossings, it is possible for the quality to depreciate rather than improve monotonically.

Finally the plots also suggest that the approximate runtime factor of two, suggested in §2.5, is fairly good. The final point on the FDP curve corresponds to $\lambda = 0.9995$ or 9,210 iterations whilst the final point on the MLFDP corresponds to $\lambda = 0.999$ or 4,604 iterations per level and, as can be seen, in all four plots these two points are fairly close together. Note that this analysis does not apply to FDP$_0$ because the initial temperature, t_0, is different.

3.3 Runtime complexity testing

From here on the tests are carried out at fixed cooling rate $\lambda = 0.9$ which we have found to be a good compromise between solution quality and runtime since it is close to the turning point in the MLFDP curves of Figures 6 & 7.

With this parameter fixed we then tested algorithmic complexity by comparing runtime against graph size using a set of 20 dual graphs, dime01, ..., dime20, ranging in size from $|V| = 1,095$ to $|V| = 224,843$ (note that some of these are used under different names in [19] and Laplace.0 $\equiv$ dime11, Laplace.2 $\equiv$ dime13, ..., Laplace.9 $\equiv$ dime20). They are somewhat unrepresentative, but of interest for complexity testing because each mesh is formed from the previous one using mesh refinement and so the underlying geometry is unchanged. Also, because they are duals of triangular meshes (as is 516 in §3.1), the average vertex degree approximately 3 which means that the number of edges for each graph scales linearly with the number of vertices. They are planar, but once again difficult for a spring-based placement method to draw because of the high variation in graph density (see §3.4 for further details). Two of them, dime01 & dime06, are also used in the mesh test suite (listed in Table 2 in §3.2), whilst another, dime20, is shown in §3.4.

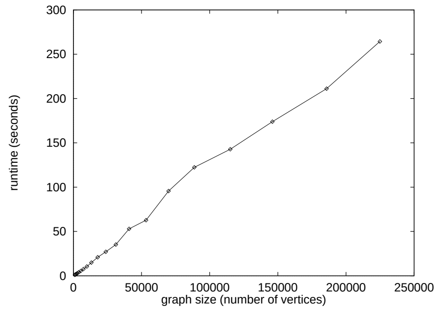


Figure 8: A plot of runtime against graph size

Figure 8 shows a plot of runtime against graph size and, as can clearly be seen, the complexity is almost linear confirming one of the conclusions in §2.5. Furthermore, the runtime for the largest graph, dime20, of nearly a quarter of a million vertices is only 264 seconds.

3.4 Further examples

In this final results section we highlight the multilevel scheme with some interesting specific layouts that it has produced (including some for graphs which have a known layout). We discuss each example graph in a little more detail in the following sections but Table 3 gives a summary in the form of a list of the graphs, their sizes ($|V|$ & $|E|$), the maximum, minimum & average degree of the vertices, the time that the multilevel algorithm required to produce a layout (and whether it was computed in 2D or 3D) and a short description. Although all of the layouts were produced automatically, for the 3D examples, included to illustrate further points, the viewpoint has been selected manually by rotating the final drawing (the choice of optimal viewpoints is itself a subject for research, e.g. [22]).

Table 3: A summary of the illustrated graphs

	size		degree			placement	
graph	$\lvert V\rvert$	$\lvert E\rvert$	max	min	avg	time (secs.)	graph type
c-fat500-10	500	46627	188	185	186.5	5.6 (2D)	random clique test
4970	4970	7400	3	2	3.0	6.4 (2D)	2D dual
4elt	15606	45878	10	3	5.9	24.3 (2D)	2D nodal
finan512	74752	261120	54	2	7.0	363.8 (2D)	linear programming
dime20	224843	336024	3	2	3.0	264.3 (2D)	2D nodal
data	2851	15093	17	3	10.6	6.6 (3D)	3D nodal
add32	4960	9462	31	1	3.8	12.5 (3D)	32-bit adder
sierpinski10	88575	177147	4	2	4.0	136.7 (3D)	2D 'fractal'
mesh100	103081	200976	4	2	3.9	431.1 (3D)	3D dual

c-fat500-10: Despite the suggestion that FDP algorithms are best suited to sparse graphs, §2.5, Figure 9(a) shows a dense regular graph (originally generated to test algorithms for the maximum clique problem). The 2D layout took around 6 seconds to compute and demonstrates that here the MLFDP algorithm has captured the symmetries nicely.

4970: The next example, Figure 10(a), is a planar dual graph derived from a mesh (also used in the mesh-based test suite, §3.2), originally constructed to highlight a problem in the mesh generator which created it. In fact by most definitions this would be considered a very poor mesh as the triangles become extremely long and thin towards the bottom left hand corner. Figure 10(b) shows the (2D) layout calculated by the MLFDP algorithm and is useful in that, by trying to equalise the edge lengths, the drawing has actually revealed far more of the graph than was originally shown. This layout took around 6 seconds to compute.

4elt: A far more challenging task for any graph-drawing algorithm which seeks to equalise edge lengths is shown in Figure 11 (also showing the detail at the centre of the mesh). This is a planar nodal graph (a larger and more complex version of the 3elt graph used in the mesh-based test suite, §3.2) which represents the fluid around a 4 element airfoil. However, because the mesh has been created to study fluid behaviour close to the airfoil, the mesh exhibits extreme variations in nodal density and a far-field (the outer border of the mesh) containing very few edges. Figure 12(a) shows the (2D) layout generated by the MLFDP algorithm and illustrates some of the difficulties. Firstly the original outer border has become the smallest of the holes whilst the outer border of the new layout is actually the perimeter of one of the original holes. Furthermore, the perimeters of the other holes exhibit buckling and folding as too many vertices have to be crammed into a space constrained in size by the rest of the graph. Possibly we could eliminate some of this folding if we increased the strength of the repulsive forces, but the layout is nonetheless fairly good and arguably shows the whole of the graph at a single resolution better than the original layout. Figure 12(b) shows some of the folding in more detail and demonstrates that the micro structure is well captured. The runtime of the MLFDP algorithm for 4elt was around 24 seconds.

finan512: Perhaps the centre-piece of these examples is taken from a linear programming matrix with around 75,000 vertices and for which no existing layout is known. Figure 13(a) shows the highly illuminating layout found by the MLFDP algorithm; the graph is revealed to have a fairly regular structure and consists of a ring with 32 'handles' each of which has a number of fronds protruding. Figure 13(b) then shows a detailed view of one of the handles. It was an extremely useful picture from the point of view of partitioning the graph because it explained why there are good natural partitions of the graph (provided that the ring is cut between the handles). This 2D layout took about 6 minutes to compute.

Note that for this particular drawing we used a 2D layout (although a 3D layout looks identical from the right viewpoint) and this example illustrates well the memory problems that can arise with the grid based simplification of repulsive forces. As explained in §2.4 this modification divides the region into square or cube shaped cells with dimensions equal to some multiple of k. If a 3D layout is chosen and the ring happens by chance to more or less align itself with one of the x, y or z axes, then a box containing the graph is relatively flat and so the number of grid cells is not unreasonable. In the worst case however, if the ring happens to lie diagonally across all three axes then the box containing the graph will be cube shaped and the number of grid cells (most of which are empty) enormous relative to $|V|$. This reinforces our suggestion of using sparse data technology to only allocate memory for non-empty grid cells.

dime20: This is the largest graph which we have tested (also included as part of the complexity testing, §3.3, and, in smaller versions, as part of the mesh-based test suite, §3.2). Once again the original mesh, Figure 14(a), exhibits extreme variations in mesh density although it is perhaps easier to draw than 4elt. The runtime to calculate the layout, shown in Figure 14(b), for this huge graph was only 264 seconds (i.e., less than $4\frac{1}{2}$ minutes).

data: This is the first 3D layout we have shown and illustrates some interesting points. Figure 15(a) shows the original nodal graph which despite the appearance of being 2D is actually a segment of the thin shell of some aeronautical body. Figure 15(b) shows the 3D layout computed by the MLFDP algorithm which, despite looking nothing like the original, demonstrates some very interesting features not least of which are the three 'panels' only weakly connected to the rest of the mesh. Until seeing this layout we had no idea of the existence of such 'panels' – the original layout certainly gives no hint of them – although they could have considerable impact on any graph-based algorithm. This layout took around 6 seconds to compute.

add32: The next graph is a representation of an electronic circuit, a 32-bit adder, for which we do not know of an existing layout. Figure 16(a) shows the results of the 3D MLFDP algorithm whilst Figure 16(b) shows a detail of the micro structure. Although the graph is not a tree (because of the existence of loops) the placement has clearly demonstrated its tree like nature with many outlying branches or fronds. The 3D layout took about 12 seconds to compute.

sierpinski10: Figure 17(a) shows the original layout of sierpinski10, a self-similar 'fractal' type structure, constructed by splitting equilateral triangles of the previous graph in the series into four (two smaller examples, sierpinski06 & sierpinski08, are used in §3.2). This is a challenging problem for the drawing algorithms because of the large holes (so that repulsive forces do not act as uniformly as in a mesh-derived graph such as 4970). Here we have chosen to draw a 3D layout, shown in Figure 17(b), despite the fact that the original graph is planar. This helps to prevent vertices from becoming trapped in local optima (as discussed by Fruchterman & Reingold, [7]) since repulsive forces need not push vertices through groups of other vertices. The 3D layout adds approximately an additional 50% time penalty (as might be expected) but the runtime is still less than $2\frac{1}{2}$ minutes. Finally note that the bottom left-hand corner is not compacted in on itself, it is merely bent backwards along the line of vision; rotating the picture reveals this corner but hides other details. It is unfortunate that the drawing procedure has not managed to map the graph to a flat plane, but with an interactive visualisation tool this does not matter too much.

mesh100: The final graph in this section is one of the largest that we have experimented with, over 100,000 vertices, and illustrates one of the problems that any graph-drawing algorithm faces. The graph is the dual of a 3D tetrahedral solid mesh and as such, with none of the face information that exists in the mesh, it is very difficult to draw meaningfully. Even in the original layout, Figure 18(a), 3D solid objects are seen to be very difficult to draw with a graph. Figure 18(b) shows the 3D MLFDP layout which took just over 7 minutes to compute. It suffers from the same problems as the original although it is splayed out because of the repulsive forces; however the symmetry is captured nicely.

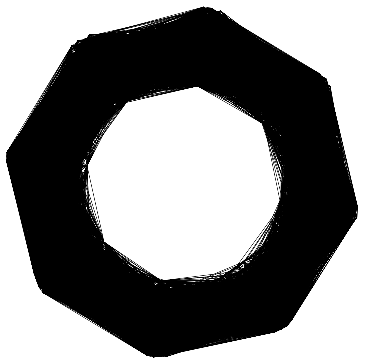
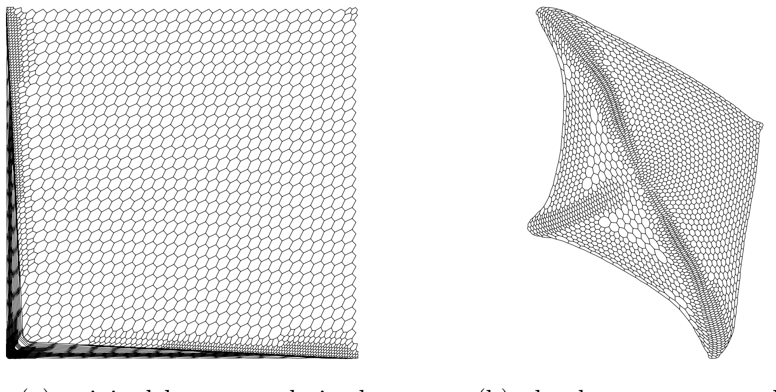


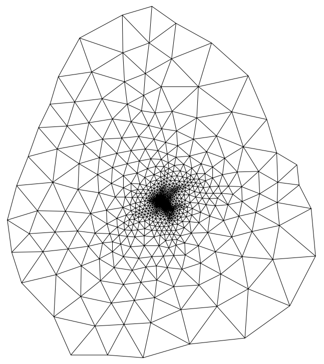
Figure 9: The graph c-fat500-10



(a) original layout as derived from the mesh

(b) the layout computed by multilevel placement

Figure 10: The graph 4970


(a) original layout as derived from the mesh

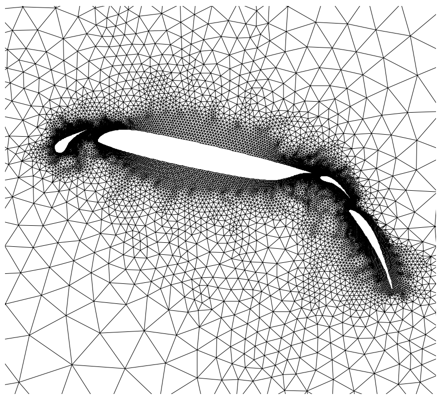
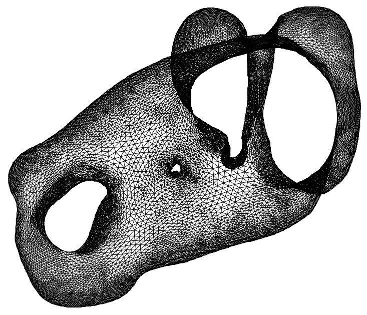

(b) detail of central region

Figure 11: The graph 4elt


(a) the layout computed by multilevel placement

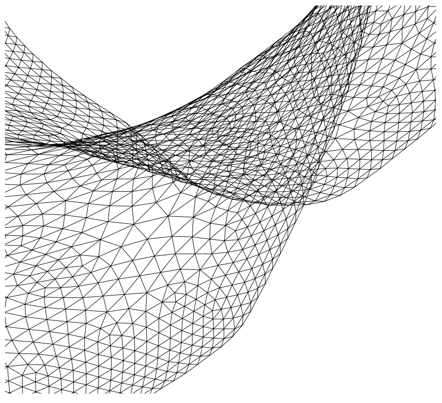
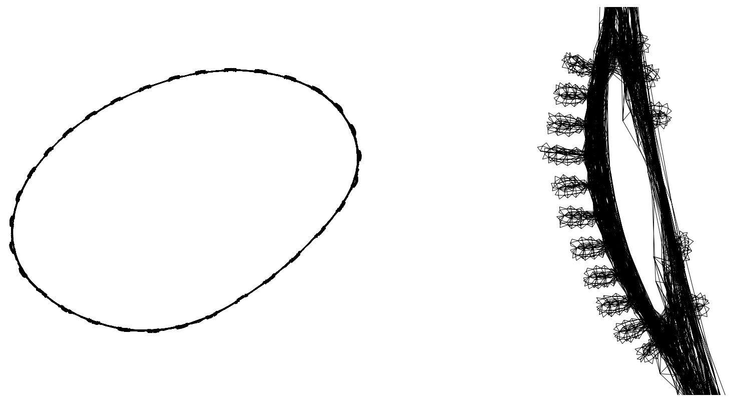

(b) detail of the folding

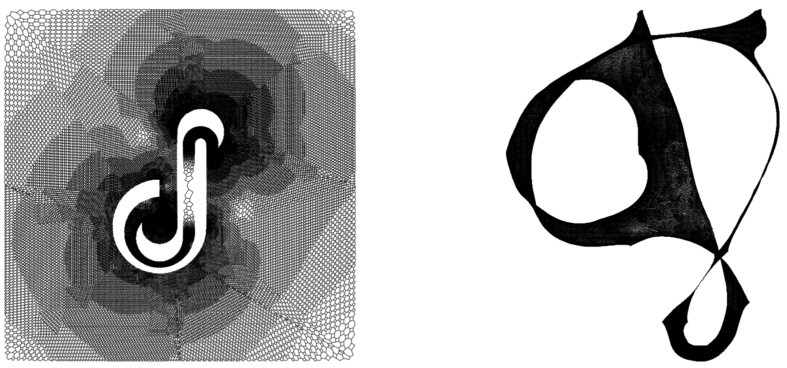
Figure 12: The graph 4elt



(a) the layout computed by multilevel placement

(b) detail of a “handle”

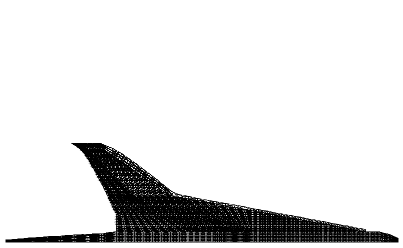
Figure 13: The graph finan512



(a) original layout as derived from the mesh

(b) the layout computed by multilevel placement

Figure 14: The graph dime20


(a) original layout as derived from the mesh

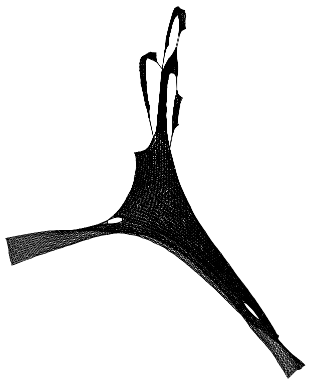
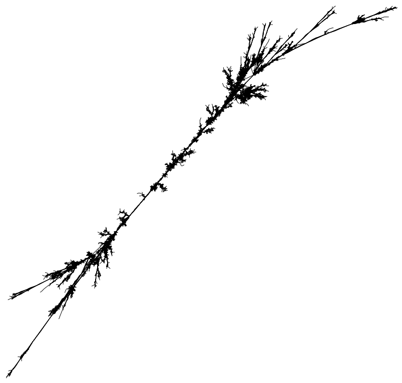

(b) the layout computed by multilevel placement

Figure 15: The graph data


(a) the layout computed by multilevel placement

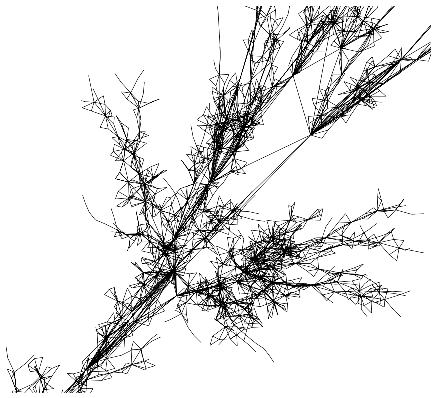
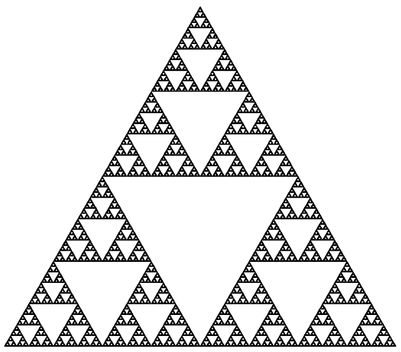

(b) the multilevel layout – detail

Figure 16: The graph add32


(a) original layout

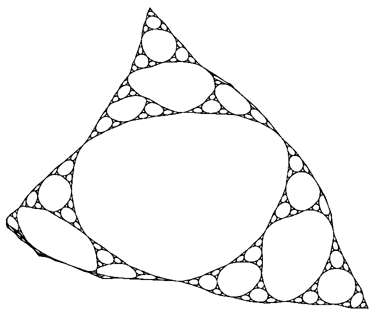
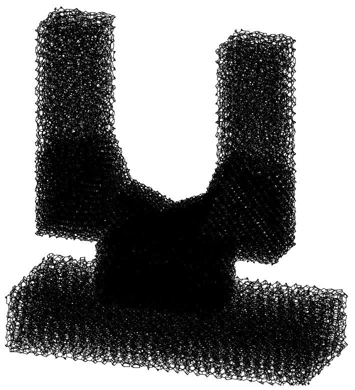

(b) the layout computed by multilevel placement

Figure 17: The graph sierpinski10


(a) original layout as derived from the mesh

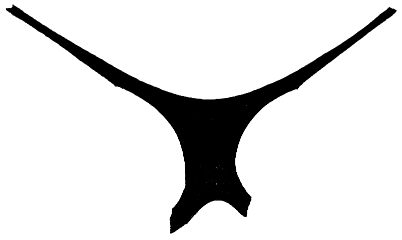

(b) the layout computed by multilevel placement

Figure 18: The graph mesh100

4 Summary and further research

We have described a multilevel algorithm for force-directed graph-drawing. The algorithm does not actually operate simultaneously on multiple levels of the graph (as, for example, a multigrid algorithm might) but instead, inspired by the multilevel partitioning paradigm, refines the layout at each level and then extends the result onto the next level down. The algorithm is fast, e.g. for sparse graphs the runtime is about 12 seconds for a 2D layout of up to around 10,000 vertices and about 5-7 minutes for 75-100,000 vertices. At the time that the algorithm was originally devised (2000) this was an order of magnitude faster than existing single-level implementations of force-directed placement (e.g. 135 seconds for a 1,000 vertex sparse planar mesh-based graph in [18, pp. 421]; even taking into account the fact that the machine used for this calculation was notionally 8 times slower this is still 16 times slower than similar examples in Table 2 which took around 1 second). It also broadens the scope of physically based graph-drawing algorithms by imparting a more global element to the layout and seems to work robustly on a range of different graphs. Furthermore, although the number of coarse graphs is typically $O(\log_2 N)$, it only adds an approximate factor of two runtime overhead to the force-directed algorithm despite considerably enhancing the results. Finally it has sometimes been suggested that it is unnecessary to draw large graphs as the human eye can not distinguish more than about 500 vertices. However examples such as finan512 in §3.4 contradict this and indicate that graphs need to be drawn at the level of the structure contained within them, although this may suggest that it is fruitless to test drawing algorithms on very large random graphs (i.e., with no structure).

So far we have tested the algorithm on a number of different graphs including several derived from unstructured meshes which tend to be relatively homogeneous in both vertex degree and local adjacency patterns. An obvious source of further research is to test the technique on graphs arising from different areas (e.g. models of social or communications networks or the internet). Our algorithm also allows vertex weights and although we have only tested this in the context of the multilevel procedure, its use with weighted graphs might provide further interesting insights. In addition we believe, partly because of our experience in dynamic repartitioning algorithms, that the multilevel process is well suited to handling dynamically changing graphs and this looks to be a fruitful topic for future research. We have not addressed disconnected graphs but feel that this requires only minor modifications. Finally we suspect that further work on some of the parameters of the algorithm would enhance its robustness and efficiency. In particular the calculation of the natural spring length k seems almost too simple to be effective.

We have not particularly tried to address graphs for which the technique might not work. It is likely that very dense graphs, or even those such as mesh100 which have a dense substructure, are never going to be good candidates for any graph-drawing algorithm, and ours is no exception. It is also likely that graphs of small diameter may not particularly suit the coarsening process (see

§2.5) although it might be possible to develop modifications to the algorithm which could deal with hubs or star graphs (e.g. by contracting the whole star in one step). In summary, however, we believe that the multilevel process can accelerate and enhance FDP algorithms for a range of useful graphs and further testing on different types of graph is an important subject for further research.

Acknowledgements

The author would like to acknowledge Yehuda Koren for interesting discussions relating to this work and thank the anonymous referees for their helpful comments.

References

[1] R. Davidson and D. Harel. Drawing Graphs Nicely using Simulated Annealing. *ACM Trans. Graphics*, 15(4):301–331, 1996.

[2] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice-Hall, Englewood Cliffs, NJ, 1998.

[3] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Planarity-Preserving Clustering and Embedding for Large Planar Graphs. In J. Kratochvíl, editor, *Proc. 7th Intl. Symp. Graph Drawing*, volume 1731 of *LNCS*. Springer, Berlin, 1999.

[4] P. Eades. A Heuristic for Graph Drawing. *Congr. Numer.*, 42:149–160, 1984.

[5] P. Eades and Q. Feng. Multilevel Visualization of Clustered Graphs. In *Proc. 4th Intl. Symp. Graph Drawing*, volume 1190 of *LNCS*, pages 101–112. Springer, Berlin, 1996.

[6] P. Eades, Q. Feng, X. Lin, and H. Nagamochi. Straight-Line Drawing Algorithms for Hierarchical Graphs and Clustered Graphs. Tech. rep. 98-03, Dept. Comp. Sci., Univ. Newcastle, Callaghan 2308, Australia, 1998.

[7] T. M. J. Fruchterman and E. M. Reingold. Graph Drawing by Force-Directed Placement. *Software — Practice & Experience*, 21(11):1129–1164, 1991.

[8] P. Gajer, M. T. Goodrich, and S. G. Kobourov. A Multi-dimensional Approach to Force-Directed Layouts of Large Graphs. In J. Marks, editor, *Graph Drawing, 8th Intl. Symp. GD 2000*, volume 1984 of *LNCS*, pages 211–221. Springer, Berlin, 2001.

[9] R. Hadany and D. Harel. A Multi-Scale Algorithm for Drawing Graphs Nicely. *Discrete Applied Mathematics*, 113(1):3–21, 2001.

[10] D. Harel and Y. Koren. A Fast Multi-Scale Algorithm for Drawing Large Graphs. *J. Graph Algorithms Appl.*, 6(3):179–202, 2002.

[11] D. Harel and Y. Koren. Graph Drawing by High-Dimensional Embedding. In S. G. Kobourov and M. T. Goodrich, editors, *Proc. Graph Drawing, 10th Intl. Symp. GD 2002*, volume 2528 of *LNCS*, pages 207–219. Springer, Berlin, 2002.

[12] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In S. Karin, editor, *Proc. Supercomputing '95, San Diego.* ACM Press, New York, 1995.

[13] T. Kamada and S. Kawai. An Algorithm for Drawing General Undirected Graphs. *Information Process. Lett.*, 31(1):7–15, 1989.

[14] Y. Koren, L. Carmel, and D. Harel. ACE: A Fast Multiscale Eigenvector Computation for Drawing Huge Graphs. In *Proc. IEEE Symp. Information Visualization (InfoVis'02)*, pages 137–144. IEEE, Piscataway, NJ, 2002.

[15] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[16] H. Purchase. Which Aesthetic has the Greatest Effect on Human Understanding? In G. Di Battista, editor, *Proc. 5th Intl. Symp. Graph Drawing*, volume 1353 of *LNCS*, pages 248–261. Springer, Berlin, 1997.

[17] R. Sablowski and A. Frick. Automatic Graph Clustering. In *Proc. 4th Intl. Symp. Graph Drawing*, volume 1190 of *LNCS*, pages 395–400. Springer, Berlin, 1996.

[18] D. Tunkelang. JIGGLE: Java Interactive General Graph Layout Environment. In S. H. Whitesides, editor, *Proc. 6th Intl. Symp. Graph Drawing*, volume 1547 of *LNCS*, pages 413–422. Springer, Berlin, 1998.

[19] C. Walshaw. A Multilevel Algorithm for Force-Directed Graph Drawing. In J. Marks, editor, *Graph Drawing, 8th Intl. Symp. GD 2000*, volume 1984 of *LNCS*, pages 171–182. Springer, Berlin, 2001.

[20] C. Walshaw. Multilevel Refinement for Combinatorial Optimisation Problems. (To appear in Annals Oper. Res.; originally published as Univ. Greenwich Tech. Rep. 01/IM/73), 2001.

[21] C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, 2000. (originally published as Univ. Greenwich Tech. Rep. 98/IM/35).

[22] R. J. Webber. *Finding the Best Viewpoints for Three-Dimensional Graph Drawings.* PhD thesis, Dept. Comp. Sci., Univ. Newcastle, Callaghan 2308, Australia, 1998.