A Multilevel Algorithm for Force-Directed Graph Drawing

C. Walshaw

School of Computing and Mathematical Sciences,
University of Greenwich, Park Row, Greenwich, London, SE10 9LS, UK.
email: C.Walshaw@gre.ac.uk

Mathematics Research Report 00/IM/60

April 20, 2000

Abstract

We describe a heuristic method for drawing graphs which uses a multilevel technique combined with a force directed placement algorithm. The multilevel process groups vertices to form *clusters*, uses the clusters to define a new graph and is repeated until the graph size falls below some threshold. The coarsest graph is then given an initial layout and the layout is successively optimised on all the graphs starting with the coarsest and ending with the original. In this way the multilevel algorithm both accelerates and gives a more global quality to the force directed placement. The algorithm can compute both 2 & 3 dimensional layouts and we demonstrate it on a number of examples ranging from 500 to 100,000 vertices. It is also very fast and can compute a layout in around 30 seconds for a 10,000 vertex graph to around 10-20 minutes for the larger graphs.

Keywords: Graph drawing, multilevel, force directed placement.

1 Introduction

Graph drawing algorithms are a basic enabling technology which is used to help with the understanding of large sets of inter-related data. By presenting data in a visual form it can often be easily digested by the user and both regular patterns and anomalies can be more easily 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.

1.1 Motivation

The motivation behind our interest in graph drawing, and in fact behind our approach to addressing the problem, both arise from our work in the field of graph partitioning. More often than not, the types of graph we wish to partition arise from computational meshes which need to be distributed amongst the processors of a parallel computer to simulate some real life process or phenomenon. It is fairly normal to have access to the coordinate information of the mesh and hence some form of coordinate system for the graph vertices (e.g. if each vertex represents a triangular or tetrahedral element one might simply position graph vertices at the centroid of that element). However, we are sometimes required to partition graphs either with no coordinate information or for which we do not have access to the coordinate information. Our initial interest in graph drawing therefore arose in trying to visualise the graph in order to help explain partitioning results and debug algorithms. In particular we are interested in:

(a) the micro structure of the graph – this can sometimes give a good indication of whether the graph is derived from a mesh or not and possibly if the local partition refinement algorithms are likely to work well;

(b) the macro structure of the graph – this can sometimes give an indication of how easy the graph is to partition in a global sense (e.g. a long thin domain may be easier to partition than a dense ball like structure).

Graph partitioning techniques have also motivated our approach to addressing the graph drawing problem. In recent years it has been recognised that an effective way of both accelerating graph partitioning algorithms and/or, perhaps more importantly, giving them a global perspective is to use multilevel techniques. The idea is to match pairs of vertices to form *clusters*, use the clusters to define a new graph and recursively iterate this procedure until the graph size falls below some threshold. The coarsest graph is then partitioned (possibly with a crude algorithm) and the partition is successively refined on all the graphs starting with the coarsest and ending with the original. This sequence of contraction followed by repeated expansion/refinement loops is known as the multilevel partitioning and has been successfully developed as a strategy for overcoming the localised nature of the Kernighan-Lin (KL), [11], and other optimisation algorithms. The multilevel partitioning paradigm was first proposed by Barnard & Simon, [1], as a method of speeding up spectral bisection and improved by both Hendrickson & Leland, [9] and Bui & Jones, [2], who generalised it to encompass local refinement algorithms. Several algorithms for carrying out the matching of vertices have been devised by Karypis & Kumar, [10].

Perhaps more interestingly in the context of graph drawing we have recently used modifications of these ideas to optimise partitions with respect to the aspect ratio or compactness of subdomains, [16], and with respect to the mapping of the subdomains onto various heterogeneous communications networks, [15] (for example to map onto a 1D array of processors it may be beneficial to slice the domain so that each subdomain, as with each processor, has only 2 neighbours). These examples are of particular interest because, although most of the modifications are made to the **local** refinement algorithms (local in the sense that they typically involve operations confined to the immediate neighbours of a vertex), the result is that the subdomains are somehow shaped in a **global** sense (i.e. either given a compact shape or shaped so that the subdomain graph matches the faster links in the processor graph). The crucial point is that by using a suitable local refinement scheme in a multilevel framework we can achieve global quality within 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 fact such ideas have already been suggested in the literature. For example, Fruchterman & Reingold, [8], suggest the possible use of 'a multigrid technique that allows whole portions of the graph to be moved' whilst Davidson & Harel, [3], suggest a multilevel approach to 'expedite the SA [simulated annealing] process'. However, we have not yet found an implementation of such ideas.

A related but somewhat different idea is that of multilevel drawings, e.g. [7]. 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 multilevel structure can then be used to aid understanding of the graph at multiple abstraction levels, [6].

2 A multilevel algorithm for graph drawing

In this section we describe how we combine a multilevel the optimisation ideas drawn from graph partitioning 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. We assume that G is connected. For any vertex v let $\Gamma(v)$ 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).

2.2 The multilevel paradigm

As previously stated, the inspiration behind our graph drawing algorithm is the multilevel partitioning paradigm. Once again, the idea is to group vertices to form *clusters*, use the clusters to define a new graph and recursively iterate this procedure until the graph size falls below some threshold. The coarsest graph is then given an initial layout and the layout is successively optimised on all the graphs starting with the coarsest and ending with the original. The algorithm does not actually operate simultaneously on multiple levels of the graph (as, for example, a multigrid algorithm might) but instead optimises the layout at each level and then interpolates the result onto the next level down.

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. [12]. Often 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. As such we seek a fast and efficient (i.e. not necessarily optimal) algorithm that is

- gradual – 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};
- uniform – we also require that the coarsening does not change the inherent properties of the graph differently between different regions.

To suit these requirements we choose (as in the partitioning algorithm) a coarsening approach known as *matching* in which vertices are matched with at most one neighbour so that clusters are thus formed of at most two vertices. 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. [13]. Unfortunately this is too slow for our purposes and, since it is not too important for the multilevel process to solve the problem optimally, we use a variant of the edge contraction heuristic proposed by Hendrickson & Leland, [9]. 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 coarser graphs as uniform as possible, and after some experimentation, 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).

The initial layout. Having constructed the series of graphs until the number of vertices in the coarsest graph is smaller than some threshold, the normal practice of the multilevel partitioning strategy is to carry out an initial partition. In terms of graph drawing we can think of this as an analogue to the initial layout. 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 and set the initial equilibrium distance to be the distance between them.

Layout interpolation. Having optimised the layout on a graph G_l, it is interpolated onto its parent G_{l-1}. The interpolation itself is a trivial matter and matched pair 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

At each level we use a force-directed placement algorithm to draw the graph, G_l, and more importantly to provide initial positions for the parent graph G_{l-1}. There has been considerable research into graph drawing and a survey can be found in [4]. 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, [5], and is constructed around 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). 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.

The particular variant of force-directed placement that we use is based on an algorithm by Fruchterman & Reingold (FR), [8], itself a variation of Eades' original algorithm. We have made a number of 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.

```
{ initialisation }
function f_g(x, w)      := begin return −Cwk²/x end
function f_l(x, d, w)   := begin return {(x − k)/d} − f_g(x, w) end
t := t_0;
Posn := NewPosn;

while (converged ≠ 1) begin
    converged := 1;

    for v ∈ V begin
        OldPosn[v] = NewPosn[v]
    end

    for v ∈ V begin
        { initialise D, the vector of displacements of v }
        D := 0;

        { calculate global (repulsive) forces }
        for u ∈ V, u ≠ v begin
            Δ := Posn[u]-Posn[v];
            D := D + (Δ/|Δ|).f_g(|Δ|, |u|);
        end

        { calculate local (spring) forces }
        for u ∈ Γ(v) begin
            Δ := Posn[u]-Posn[v];
            D := D + (Δ/|Δ|).f_l(|Δ|, |Γ(v)|, |u|);
        end

        { reposition v }
        NewPosn[v] = NewPosn[v] + (D/|D|).min(t, |D|);
        Δ := NewPosn[v]-OldPosn[v];
        if (|Δ| > k×tol) converged := 0;
    end

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

Figure 1: Force-directed placement algorithm

Figure 1 shows the basic outline of our algorithm and is written in a similar fashion to the original FR algorithm, [8]. There are two main differences (apart from the choice of parameters); the local forces which are both attractive & repulsive, and the order of updating. We discuss these in more detail below. One other

fairly minor difference is that we do not impose any boundaries around the drawing; 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.

Updating. An important difference from the original FR algorithm is the order of updating of the vertex positions. The original algorithm used two vectors, 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 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.

At first it might seem as if our version if less efficient as the local 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 speeds up the convergence of the algorithm. 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.

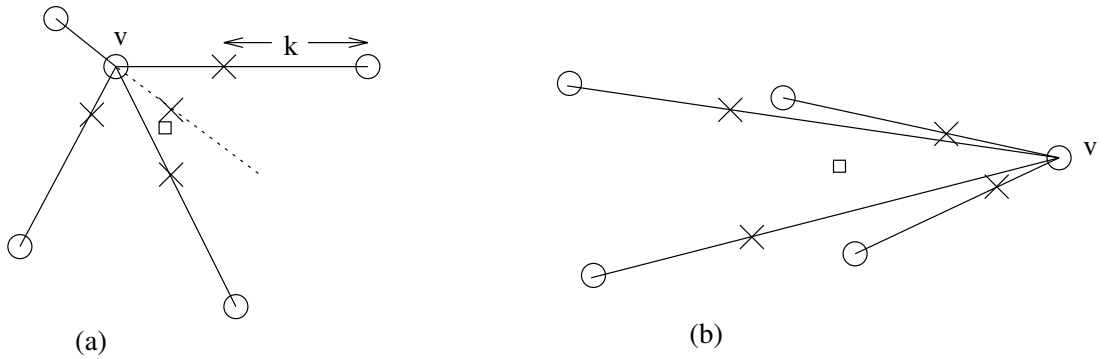

Figure 2: Calculation of displacement due to local forces

Local (spring) forces. Rather than using purely attractive local forces we use spring forces locally or perhaps more accurately spring placement. Consider the system shown in Figure 2(a) where only v is allowed to move and the other vertices (shown as circles) are fixed. The force on v from any one adjacent vertex, u say, would move it along the line passing through v and u either away from u if the spring were compressed shorter than its natural length k or towards u if the spring were overextended. Depending on the damping in the system v would eventually end up in an equilibrium position at a distance k from u along this line. The aggregate motion implied by all adjacent vertices therefore suggests moving v to the centroid (marked with a square) of the equilibria (all marked with crosses) although note that this may not in itself be a natural equilibrium position, see for example Figure 2(b). If x_u is the distance of u from v and $|\Gamma(v)|$ the degree of v (since $\Gamma(v)$ is the set of vertices adjacent to v) then the contribution to this motion due to the edge (u, v) is $(x_u - k)/|\Gamma(v)|$ in the direction of u. This is somewhat different from the original FR algorithm (and indeed its predecessor due to Eades) since it means that the local forces contain an element of repulsion; the original versions only used attractive local forces.

Note that in Figure 1 we also subtract the global repulsive force, $f_g(x_u, |u|)$, from u on v. This has the effect of cancelling out the global forces calculated earlier in the outer loop and means that global forces do not act between adjacent vertices. We offer no justification for this in the spring based analogy other than to say that since the local forces include an element of repulsion we do not feel that it is necessary and that it seems to work well in practice; certainly better than including it. It is also the strategy used by Eades in his original version of the spring embedder algorithm, [5]. An alternative implementation to neglect this force would be, during the double loop over vertices in the calculation of the global forces, to check whether every pair of vertices, (u, v) are adjacent and, if so, ignore the repulsion at this point. However we feel that this would be more time consuming than adding it in and then subtracting it later.

Global (repulsive) forces. In the original FR algorithm repulsive forces are calculated between every

pair of vertices in the graph (and thus the system resembles an n-body problem). In this sense, 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 system 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.

That the repulsive forces are necessary is therefore undoubted, but as for the actual function used to calculate them, with no physical analogue, the choice is not obvious. We have chosen, after a certain amount of experimentation, to use a slightly modified version of the FR original repulsive function $f_r(x, k) = k^2/x$ which we relabel f_g (for global). We suspect however that there may be many suitable functions which achieve similar effects.

We modify the function in two ways, firstly by multiplying by a constant C which seems to work well if chosen from the interval $[0.01, 0.1]$. The motivation behind this is that in the original the forces are enormous and unstable. Indeed it is only the fact that the movement of any vertex is constrained temperature which restricts the original FR from exploding into chaotic behaviour.

Note that we also multiply the repulsive force by the weight, $|u|$, of the vertex, u, which generates it. Although we typically (but not exclusively) are 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.

Note that there is no weighting analogue with the local spring forces except perhaps to take into account the weight of the edge between two vertices. However, we have not tested this.

Natural spring length, k. A crucial part of the algorithm 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 (except for the coarsest graph) the vertices will all be in positions determined by the layout calculated for graph G_{l+1}. 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.

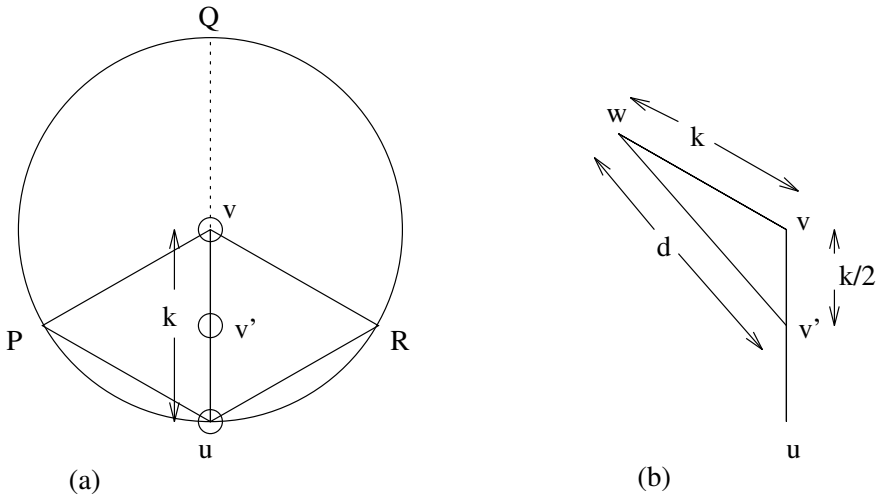

Figure 3: Calculation of natural spring length

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 k 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

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

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

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

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

Remarkably this simple formula works very robustly over all the examples that we have tested, certainly better than other functions we have tried. 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 and the algorithm converges after one iteration (because neither vertex will move). However, there are times, for example when redrawing a graph, when the coarsening is not required to reach this threshold and so the algorithm is not modified for the special case $|V_L| = 2$.

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 based on simulated annealing and due to Davidson & Harel, [3], 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). The algorithm is then assessed as having converged when the movement of every vertex is less than some tolerance, *tol*, times k. This allows us to explicitly avoid setting any maximum number of iterations although there is an implicit limit since eventually the temperature will drop below *tol*$\times k$.

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 this case the vertices are simply treated as if they were a small distance apart (the actual direction generated randomly with the distance no more than $0.001 \times k$) and the forces calculated accordingly. This allows us to interpolate the layout of one graph onto its parent without any additional sophisticated mechanism.

Other parameters. We have discussed most of the important parameters & functions above. For the examples shown in Section 3 the other parameters of the algorithm were set as follows:

function $cool(t) :=$ **begin return** $0.95t$ **end**
$t_0 := k$;
$C := 0.01$;
tol $:= 0.001$;

2.4 Reducing the complexity

It is fairly clear from the description of the algorithm that the placement complexity for each graph level, $G_l(V_l, E_l)$, is $O(|V_l|^2 + |E_l|)$. For the types of sparse graphs that we are interested in the $|V_l|^2$ heavily dominates this expression and we therefore use the FR grid variant for reducing the run-times. Their motivation was that over long distances the repulsive forces are sufficiently small enough 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 $f_g(x, w) :=$
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 if we divide the domain into regular square cells (or cube shaped cells in 3D) of size R^2 (or R^3 in 3D) we know that 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 should not be too difficult to deal with by using sparse data structure technology such as only creating the linked list header information for non-empty cells and storing these in a tree structure rather than 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 that we are interested in 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. For the initial coarse graphs we can set R to be relatively large and achieve some impressive untangling without too much cost (since $|V|$ 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 relatively small without penalising the placement. In fact the first such schedule that we tried with $R = 2(l + 1)k$ for each graph G_l worked so well that we have not experimented further. In this expression l is just the graph number where G_0 is the original graph and G_l the graph after l coarsenings. Interesting this also replicates the choice of $R = 2k$ for G_0 in the original FR algorithm.

2.5 Graph redrawing

We have described our algorithm as if positioning the vertices from scratch, but one of the attractive features of spring embedder type algorithms is that they can make use of existing layouts and (hopefully) improve them. Our algorithm is no exception although two minor issues must be addressed. Firstly the coarsening will start with an existing layout and so when any two vertices are clustered together the cluster should be positioned at the mid-point between the vertices.

More importantly, the coarsening threshold must be changed. If the graph is coarsened to 2 vertices then the algorithm will more or less proceed as if starting from scratch and all existing information will be lost (since the new layout is completely dependent on the initial layout). If, however, the coarsening stops after the coarsest graph size falls below some threshold T then a certain amount of information will be retained. There is no automatic method of choosing T and to some extent it must depend on how 'good' the existing layout is (something which is very difficult to quantify); the better the layout the lower the threshold.

Interestingly this has an exact analogue in multilevel graph partitioning when an existing but perhaps suboptimal partition needs to be repartitioned (for example to load-balance a dynamically changing mesh). The more the graph is coarsened the better the quality of the partition is likely to be. However, if the existing partition is already of high quality, the more coarsening takes place, the more the partition will change and as a result, in a parallel simulation, the more data will need to migrate from one processor to another, [14]. This reflects the 'globality' of the multilevel paradigm; more coarsening implies a more global aspect to the solution but may move the new solution further away from the previous solution.

We have not tested the redrawing facility but it is implemented and the user can set the threshold T at run time.

3 Examples

We have implemented the algorithms described here within the framework of JOSTLE, a mesh partitioning software tool developed at the University of Greenwich and freely available for academic and research purposes under a licensing agreement[1]. The experiments were carried out on a Sun SPARC Ultra 10 with a 333 MHz CPU and 256 Mbytes of memory.

We have tested our multilevel algorithm on a number of example graphs, including some for which we already know a good layout (although interestingly, even the results on these graphs proved very illuminating). Most such graphs were drawn from genuine examples of meshes from various computational mechanics problems. Typically in such graphs the vertices can either represent mesh nodes (the nodal graph), mesh elements (the dual graph), a combination of both (the full or combined graph) or some other special purpose representation. However we have also considered graphs from other non mesh-based applications.

	size		degree			placement	
graph	$\|V\|$	$\|E\|$	max	min	avg	time (secs.)	type
516	516	729	3	1	2.83	1.14	2D dual graph
data	2851	15093	17	3	10.59	20.05	3D nodal graph
add32	4960	9462	31	1	3.82	48.09	32-bit adder (electronic circuit matrix)
4970	4970	7400	3	2	2.98	13.93	2D dual graph
whitaker3	9800	28989	8	3	5.92	24.10	2D nodal graph
4elt	15606	45878	10	3	5.88	44.93	2D nodal graph
finan512	74752	261120	54	2	6.99	688.46	linear programming matrix
mesh100	103081	200976	4	2	3.90	1245.43	3D dual graph

Table 1: A summary of the example graphs

We discuss the graphs in a little more detail in the following sections but Table 1 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 a short description.

Note that all the figures shown to demonstrate the multilevel placement algorithm were calculated entirely automatically with the exception of either one or two user inputs. Firstly the user must select whether to calculate a 2D or a 3D layout and we show the most appropriate here. The second choice can occur if a 3D layout is selected and the user may then need to rotate the final drawing in order to select the best viewpoint (the choice of optimal viewpoints is itself a subject for research, e.g. [17]).

3.1 An extended example

In this section we demonstrate in a little more detail how the multilevel placement (MLP) algorithm works. Figure 4 shows the original layout, as derived from the mesh, for the smallest of our test graphs 516 (with 516 vertices). It also shows (lightly shaded) the underlying triangular mesh. In this case the graph is a so-called dual graph where each vertex represent a triangular element.

The MLP algorithm is run on this graph (ignoring the existing layout) and Table 2 list the sizes of the coarsened graphs, $G_1(V_1, E_1) - G_9(V_9, E_9)$ which are constructed by the coarsening. Notice that $|V_l| \geq$

[1]available from `http://www.gre.ac.uk/jostle`

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

Table 2: The sizes of the coarsened graphs of 516

$|V_{l-1}|/2$ 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 interpolated from G_{l+1}, by simply placing vertices at the same position as the cluster representing them in the coarser graph, and then optimised.

Figure 5 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 6-8 meanwhile illustrate the placement algorithm on G_2. Figure 6 shows the initial layout as calculated on G_3 and with many of the vertices coincident whilst Figure 7 then shows the layout after the first iteration and where the coincident vertices have started to separate. Figure 8 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. Figure 9 shows the final layout on the original graph, G_0. The slight kink arises from the hole in the graph which distorts the layout when vertices clustered over it, but in general the final layout is very good and is certainly sufficient to suggest (to an experienced eye) that the graph arises from the dual of a triangular mesh. Finally note that the MLP algorithm took just over 1 second to compute this layout.

For comparison, Figure 10 shows the placement algorithm used on a random initial layout of 516 (in other words as a standard single level placement algorithm). Possibly the algorithm is not well tuned for this problem although we have raised the initial temperature to $10k$), but what is see is that although the micro structure of has been reconstructed reasonably well (at least this can be seen by examining the layout in more detail than Figure 10 allows), the single level placement has not been able to 'untangle' the graph in a global sense. We believe that this at least hints at the global power of the multilevel algorithm.

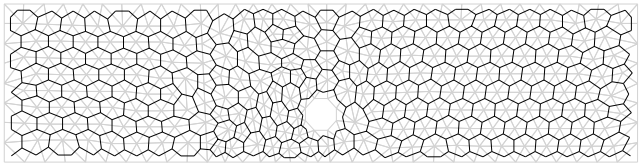

Figure 4: The original layout of 516 as derived from the mesh also showing the undlying triangular elements

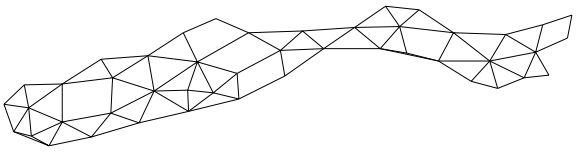

Figure 5: The final layout for G_4 of the 516 graph

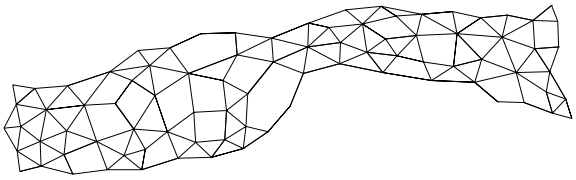

Figure 6: The initial layout for G_2 of the 516 graph

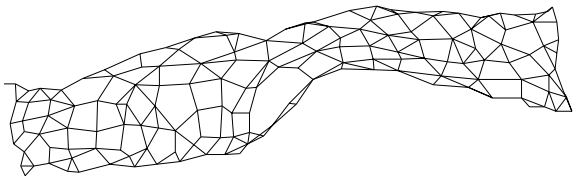

Figure 7: The layout after the first iteration for G_2 of the 516 graph

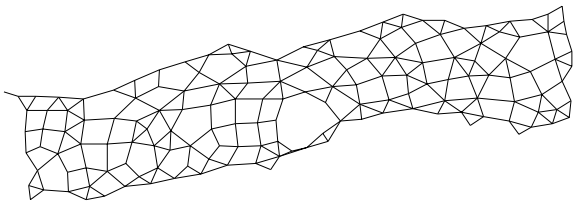

Figure 8: The final layout for G_2 of the 516 graph

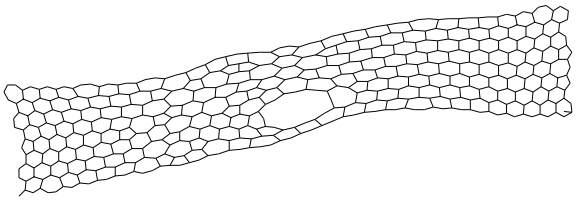

Figure 9: The final layout of 516 computed with the multilevel placement algorithm

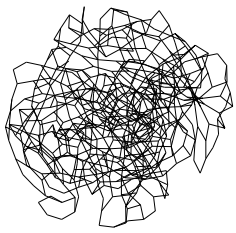

Figure 10: The final layout of 516 computed with single level placement

3.2 Graphs with known layouts

In this section we present some more difficult examples although again we have an existing layout to compare the results against.

whitaker3: The first example is once again a planar graph derived from a mesh (although this time the vertices represent mesh nodes); however it is a much larger example with nearly 10,000 vertices. Figure 11 shows the original layout and we can see that, happily for the placement algorithm, the edges are all of very similar length. Figure 12 meanwhile shows the (2D) layout achieved by the MLP algorithm and we see that, apart from some slight stretching, it has done a very good job. Furthermore, this layout was calculated in less than 25 seconds.

4970: A slightly more challenging graph is shown in Figure 13. This is the dual of a triangular mesh which was 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 14 shows the (2D) layout calculated by the MLP algorithm and is impressive in that, in trying to equalise the edge lengths, the drawing has actually revealed far more of the graph than was originally shown. This layout took around 14 seconds to compute.

4elt: A far more challenging task for any graph drawing algorithm which seeks to equalise edge lengths is shown in Figures 15 & 16 (this latter showing the detail at the centre of the mesh). Once again this is a 2D planar nodal graph which in fact represents the fluid around a 4 element airfoil but, because the mesh has been created to study fluid behaviour close the 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 17 shows the (2D) layout generated by the MLP 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 mesh. 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 18 shows some of the folding in more detail but demonstrates that the micro structure is very good. The runtime of the MLP algorithm for 4elt was around 45 seconds.

data: This is the first 3D graph we have shown and illustrates some interesting points. Figure 19 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 20 shows the 3D layout computed by the MLP 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, say, the partition of the graph. This layout took around 20 seconds to compute.

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 21, 3D solid objects are seen to be very difficult to draw with a graph. Figure 22 shows the 3D MLP layout which took about 21 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.

3.3 Graphs without known layouts

In this section we look at two example graphs[2] for which we do not have an existing layout.

add32: The first graph is a representation of an electronic circuit, a 32-bit adder. Figure 23 shows the results of the 3D MLP algorithm whilst Figure 24 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. It was an extremely useful picture from the point of view of partitioning the graph because it explained why the graph allows some (easy to find) natural partitions. The layout took under 50 seconds to compute.

[2]both available in the form of matrices from the Florida sparse matrix collection `ftp://ftp.cis.ufl.edu/pub/umfpack/matrices/`

finan512: The final example graph is taken from a linear programming matrix. Figure 25 shows the layout found by the MLP algorithm and Figure 26 shows a detailed view of one of the 'handles'. Once again this layout is highly illuminating; the graph is revealed to have a fairly regular structure and consist of a ring with 32 handles each of which has a number of fronds protruding. Again this explains why there are good natural partitions of the graph provided that the ring is cut between the handles. This layout took less than 12 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.

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, optimises the layout at each level and then interpolates the result onto the next level down. The algorithm is fast, e.g. about a second for a 500 vertex graph, under 30 seconds for a 2D layout of up to around 10,000 vertices and about 10 to 20 minutes for 75-100,000 vertices. It also seems to work robustly on a range of different graphs.

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. However it is also possible that graphs containing vertices of very high degree may not particularly suit the multilevel process. We have not dealt with disconnected graphs but feel that this requires only a few minor modifications.

So far we have tested the algorithm on a number of different graphs mostly drawn from our a suite of examples assembled to test graph partitioning algorithms. Many of these derive from unstructured meshes and as such are 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 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.

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. In addition, we feel that better investigation of the repulsive forces might alleviate some of the (minor) problems demonstrated in the examples. On a larger scale we would also be interested in investigating techniques for which k does not have to be constant but can vary throughout the graph.

References

[1] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice & Experience*, 6(2):101–117, 1994.

[2] T. N. Bui and C. Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In R. F. Sincovec *et al.*, editor, *Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.

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

[4] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for Drawing Graphs: An Annotated Bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.

[5] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1984.

[6] P. Eades and Q. Feng. Multilevel Visualization of Clustered Graphs. In *Proc. 6th Int. Symp. Graph Drawing*, volume 1190 of *LNCS*, pages 101–112. Springer, 1996. (Berkeley, California).

[7] 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.

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

[9] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In S. Karin, editor, *Proc. Supercomputing '95*, New York, NY 10036, 1995. ACM Press.

[10] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[11] B. W. Kernighan and S. Lin. An Efficient Heuristic for Partitioning Graphs. *Bell Systems Tech. J.*, 49:291–308, February 1970.

[12] J. Kratochvíl, editor. *Proc. 7th Int. Symp. Graph Drawing, GD '99, Stirin Castle, Czech Republic*, volume 1731 of *LNCS*. Springer, 1999.

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

[14] C. Walshaw and M. Cross. Load-balancing for parallel adaptive unstructured meshes. In M. Cross *et al.*, editor, *Proc. Numerical Grid Generation in Computational Field Simulations*, pages 781–790. ISGG, Mississippi, 1998.

[15] C. Walshaw and M. Cross. Multilevel Mesh Partitioning for Heterogeneous Communication Networks. Tech. Rep. 00/IM/57, Univ. Greenwich, London SE10 9LS, UK, March 2000.

[16] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel Mesh Partitioning for Optimising Domain Shape. *Int. J. High Performance Comput. Appl.*, 13(4):334–353, 1999. (originally published as Univ. Greenwich Tech. Rep. 98/IM/38).

[17] R. J. Webber. *Finding the Best Viewpoints for Three-Dimensional Graph Drawings*. PhD thesis, Dept. Comp. Sci. & Software Engrg., Univ. Newcastle, Australia, 1998.

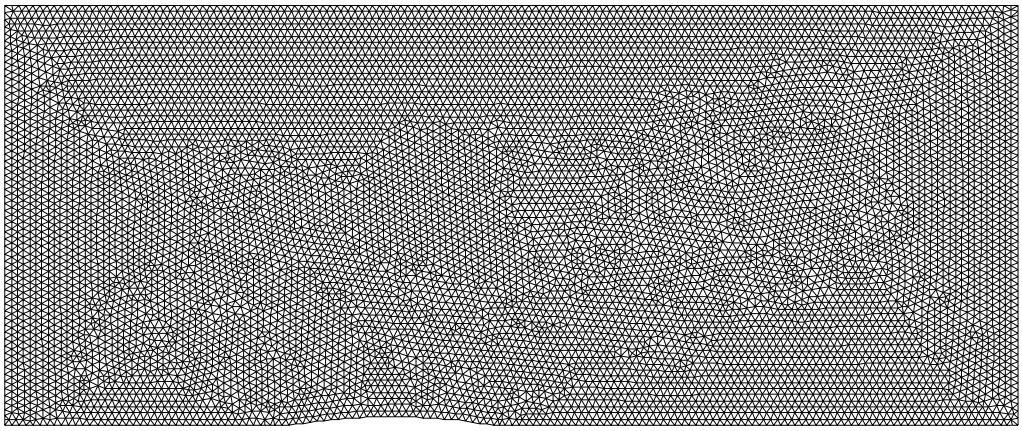

Figure 11: The original layout of whitaker3 as derived from the mesh

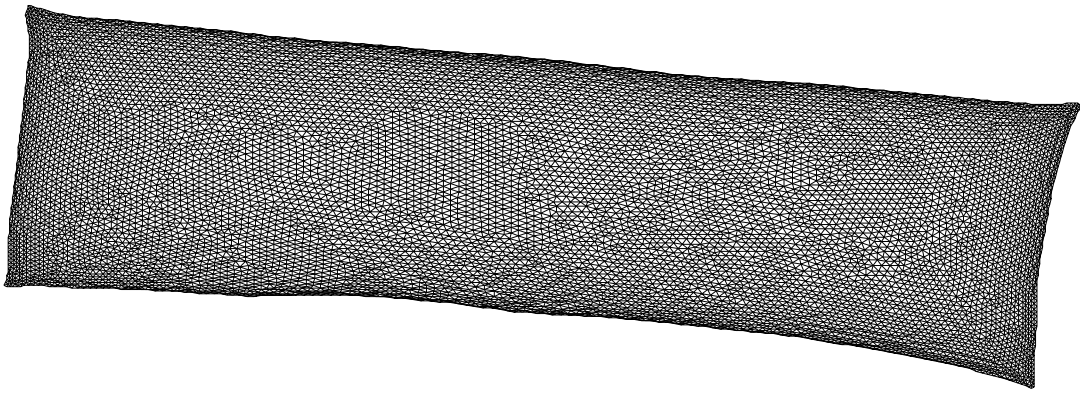

Figure 12: The layout of whitaker3 computed with the multilevel placement algorithm

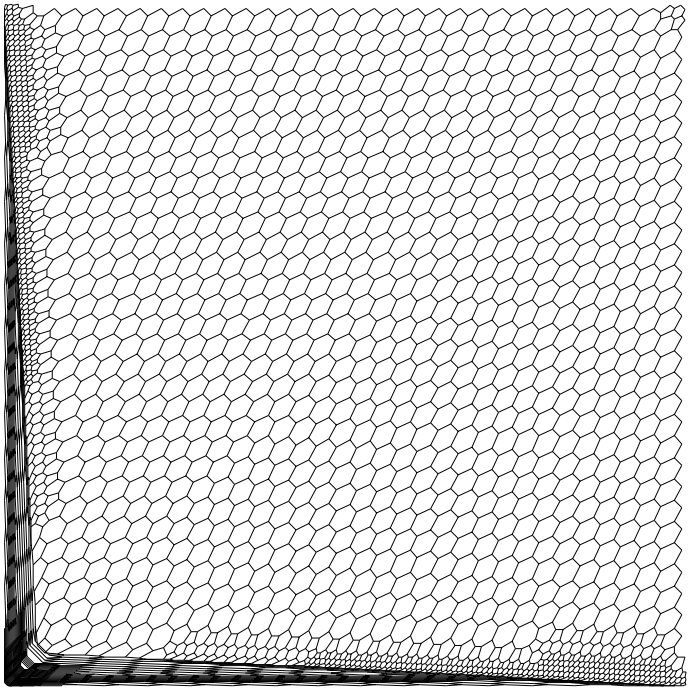

Figure 13: The original layout of 4970 as derived from the mesh

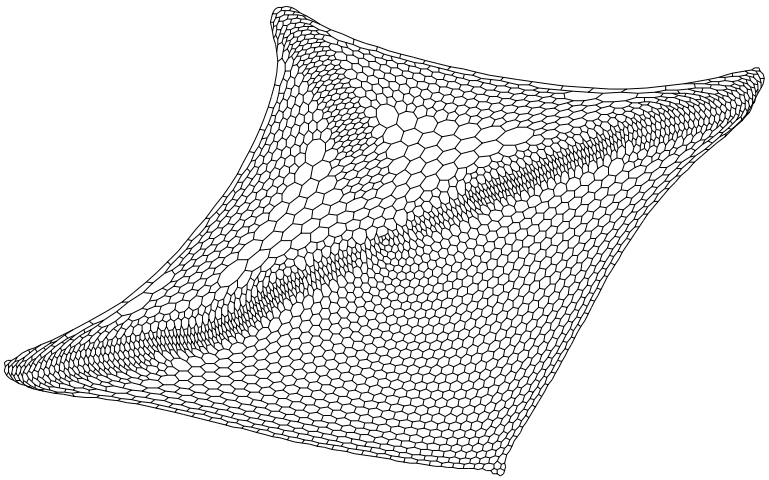

Figure 14: The layout of 4970 computed with the multilevel placement algorithm

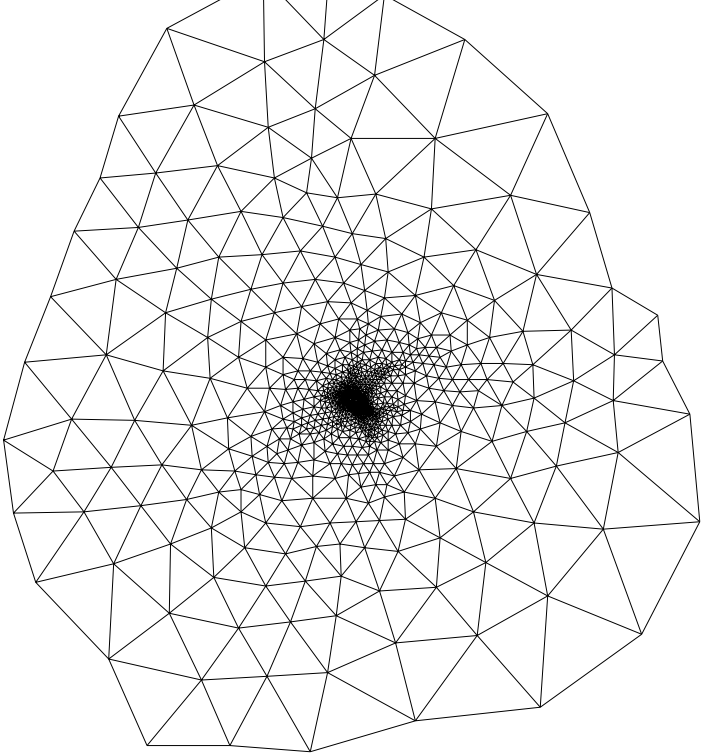

Figure 15: The original layout of 4elt as derived from the mesh

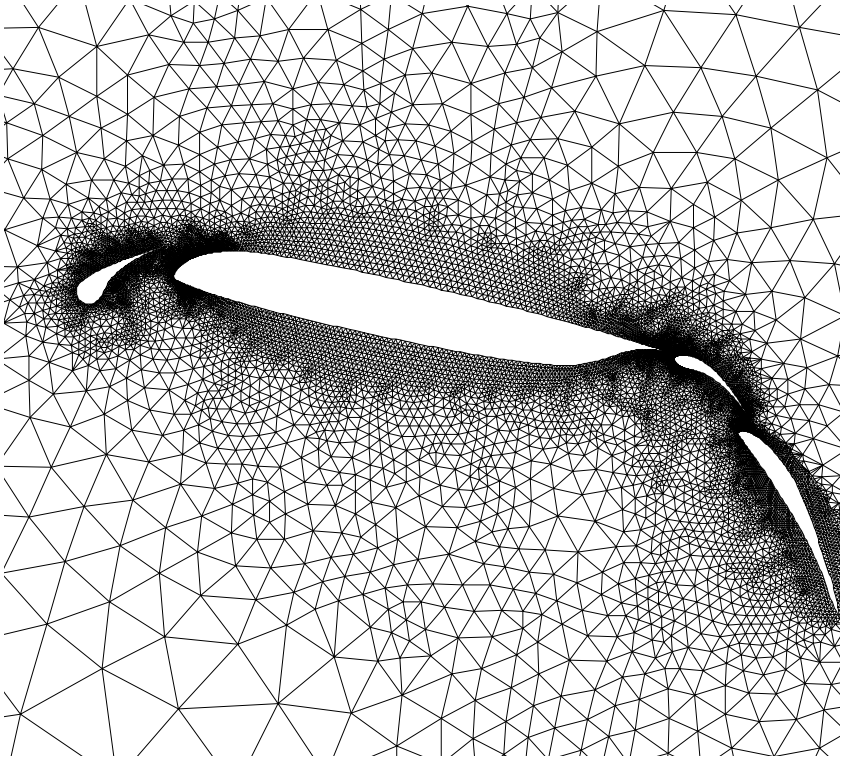

Figure 16: 4elt as derived from the mesh: detail of the mesh centre

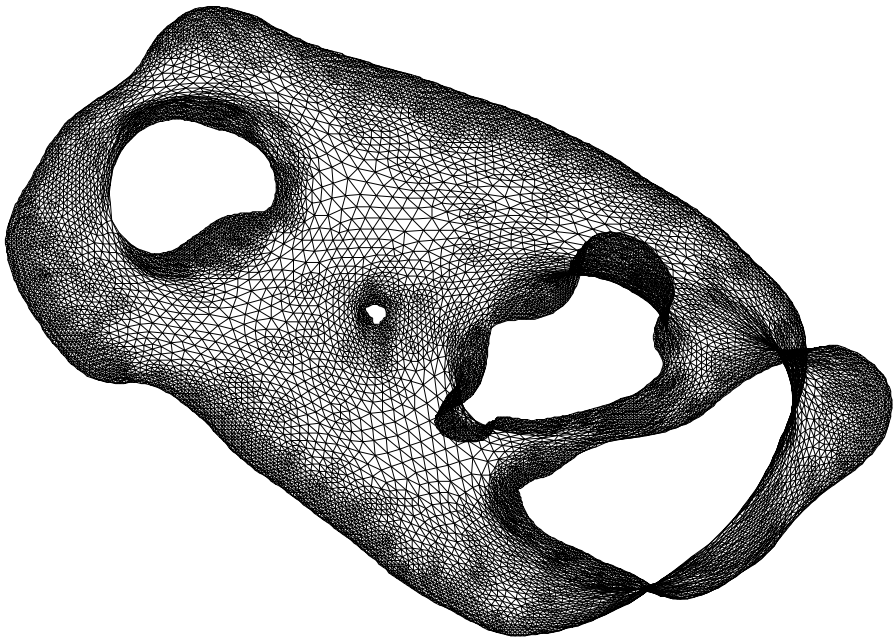

Figure 17: The layout of 4elt computed with the multilevel placement algorithm

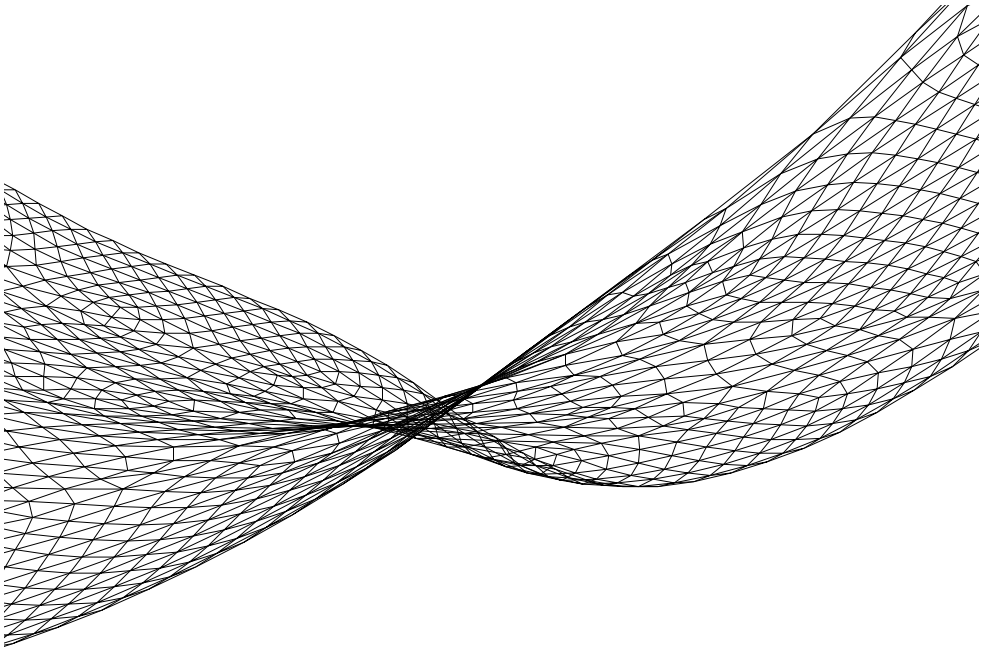

Figure 18: 4elt computed with the multilevel placement algorithm: detail of the folding

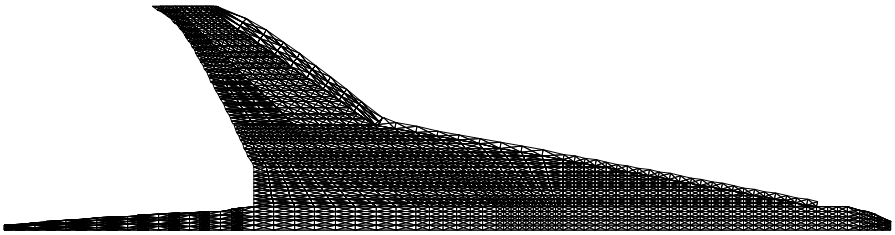

Figure 19: The original layout of data as derived from the mesh

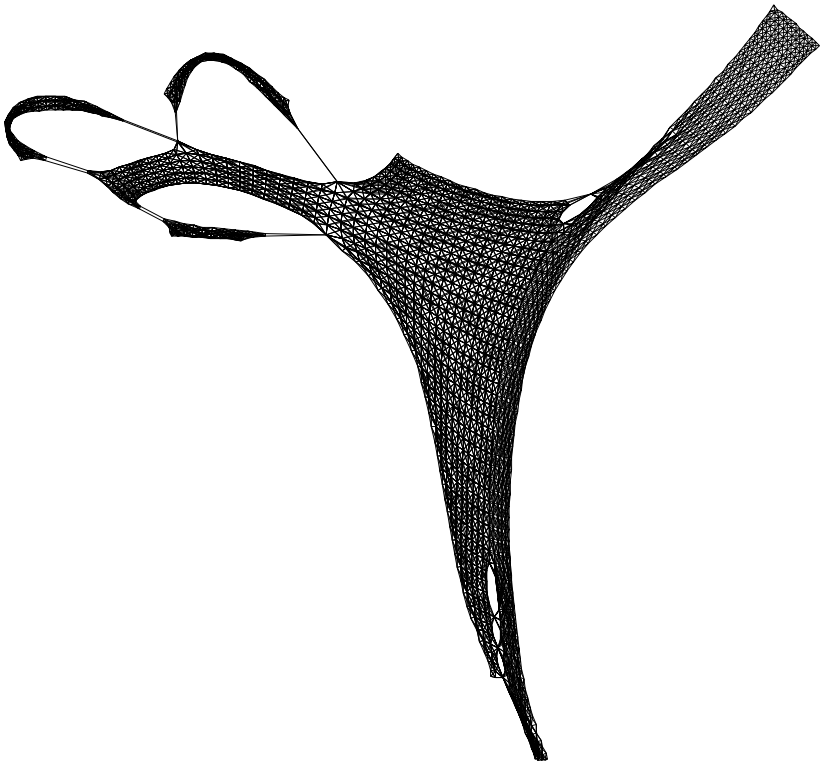

Figure 20: The layout of data computed with the multilevel placement algorithm

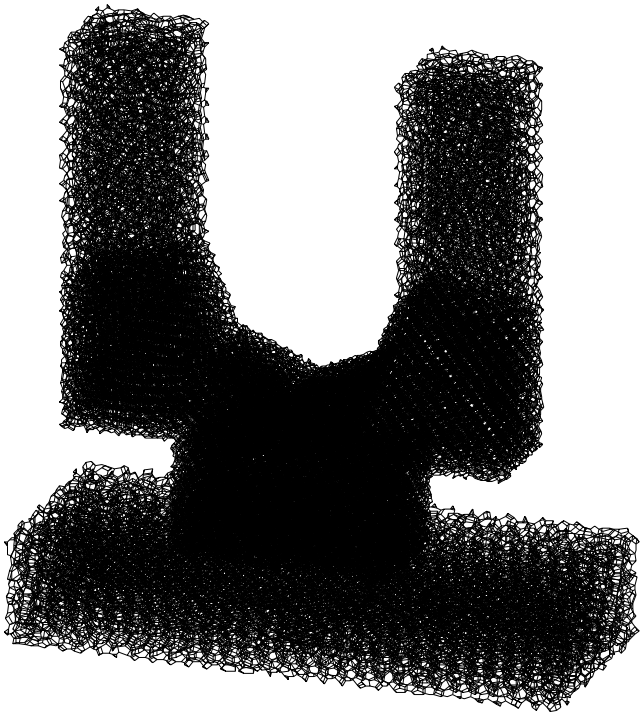

Figure 21: The original layout of mesh100 as derived from the mesh

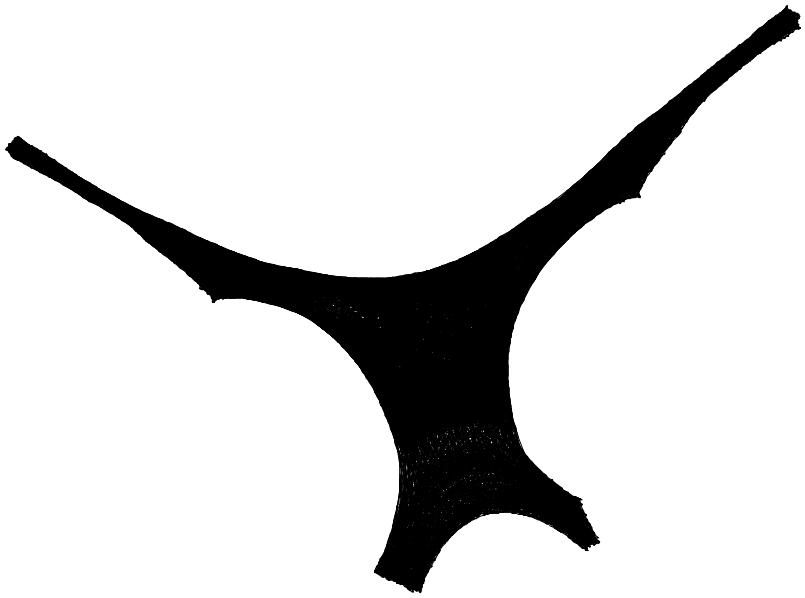

Figure 22: The layout of mesh100 computed with the multilevel placement algorithm

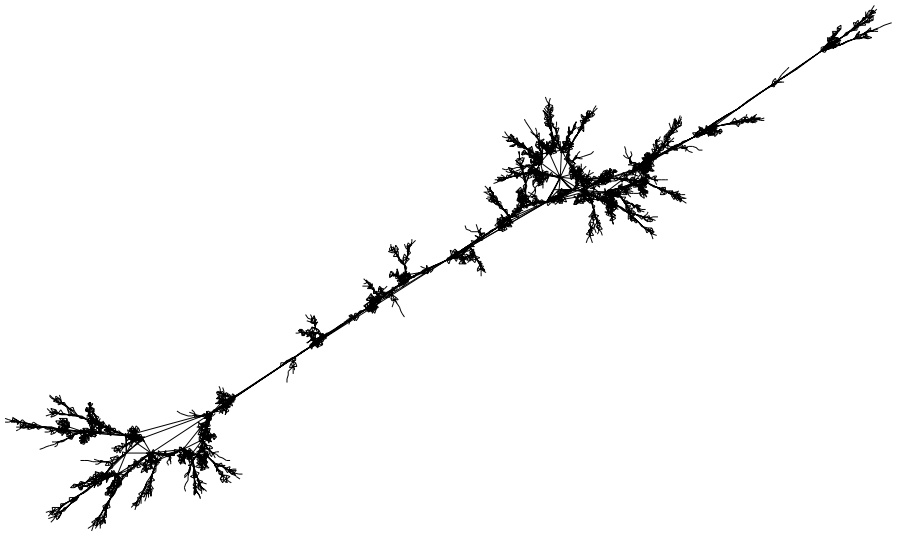

Figure 23: The layout of add32 computed with the multilevel placement algorithm

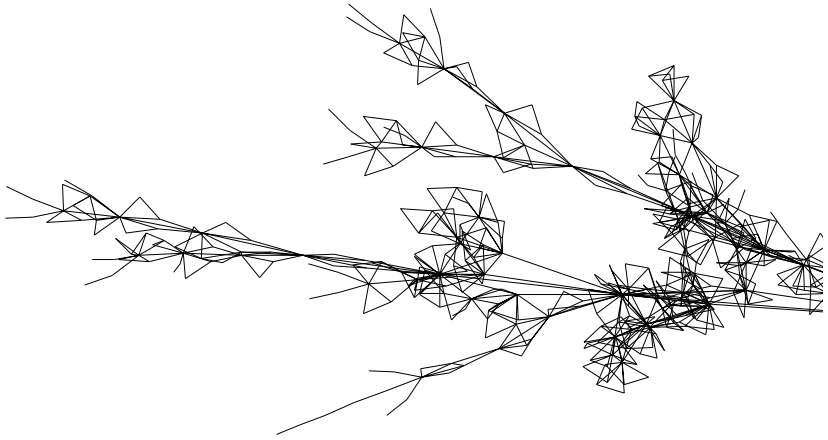

Figure 24: add32 computed with the multilevel placement algorithm: detail of the micro structure

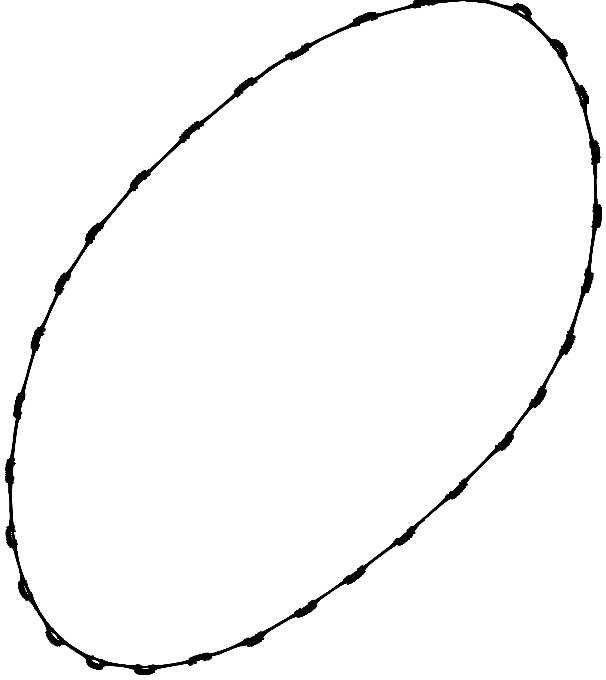

Figure 25: The layout of finan512 computed with the multilevel placement algorithm

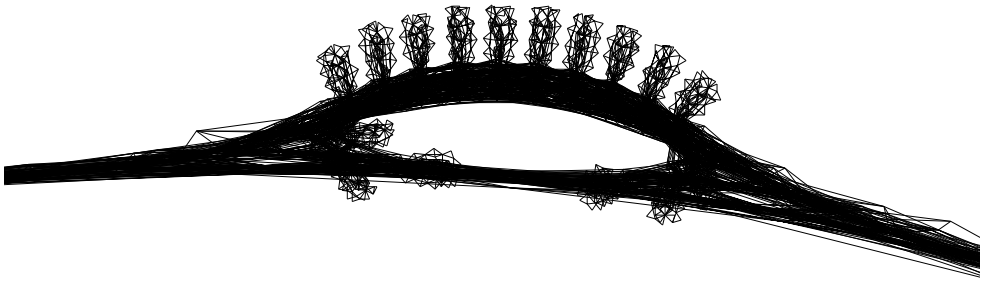

Figure 26: finan512 computed with the multilevel placement algorithm: detail of the micro structure