Spring Embedder Preprocessing for WWW Visualization

Paul Mutton, Peter Rodgers
University of Kent at Canterbury
{pjm2@ukc.ac.uk, P.J.Rodgers@ukc.ac.uk}

Abstract

We present a preprocessor for the spring embedder graph drawing method and show its use in speeding up the automatic layout of three-dimensional visualizations of WWW sites. Spring embedding is a widely used method for visualizing the connections in WWW maps, as it can typically produce a reasonable layout for most general graphs. However, the technique does not scale well and to improve the performance when dealing with large graphs various optimisations have been developed. Our preprocessor is a new optimization method that attempts to obtain a reasonably good initial drawing to be then used by the spring embedder. This initial drawing has edge lengths that are approximately equal along with a minimum node separation. This produces a layout that is closer to the final drawing than a random scattering of nodes and so allows fewer invocations of the spring embedder to produce an equally stable drawing.

1. Introduction

WWW sites are often visualized by graphs, with nodes representing pages and edges representing hyperlinks between pages. Graph drawing methods are important in such visualizations for automatically laying out the site in an often aesthetically pleasing manner. A good layout can ease user exploration when discovering paths between pages or highlighting clusters of pages within the site. The use of three dimensions allows large sites to be navigated more effectively than using only two-dimensions, with advantageous usability issues in spatial navigation, layout and semiotics [8,9,14].

Spring embedding [2,4] is a technique used for the drawing of such graphs [6,7]. Its effect is to distribute nodes with some separation, whilst attempting to keep connected nodes reasonably close together. In order to speed up the spring embedder, we have developed a novel two stage preprocessor that attempts to get a first rough cut at node distribution whilst keeping nearby nodes at a suitably close distance. This allows a much quicker equilibrium to be reached by the spring embedding that follows. In a few cases, these initial drawings do not require any subsequent spring embedding to achieve a good layout. This particular aspect of the pre-processor is best observed on mesh-like graph structures and graphs containing several distinct clusters, where it is common to have many similar edge lengths.

The spring embedder graph drawing process considers the graph model as a force system that must be simulated. Nodes are charged particles that repulse each other and edges are springs that attract two nodes together. The graph is drawn by repeated iterations of a procedure that calculates the repulsive and attractive forces on all the nodes in the graph, and moves the nodes accordingly. The drawing process stops after a predetermined number of iterations, or when the nodes stop moving, which implies an equilibrium has been found. Much previous work has concentrated on optimising the spring embedder by speeding up the calculation of forces between pairs of nodes [12] or reducing the number of nodes that are paired [11,12]. Multi-level [5,13] approaches provide a heuristic method that clusters a graph and lays out the coarsened graph, reintroducing the other nodes in uncoarsening steps until a final drawing is produced. Other heuristics, for example those provided by the GEM system [1], can detect oscillations of nodes and rotations of subgraphs. However, most variations of the traditional spring embedder commonly start by being applied to a graph where the nodes have been allocated to random locations, a binary tree or a mesh [3]. Very little research has been performed on optimising spring embedding by considering this initial layout.

Our preprocessor produces a reasonable initial layout before the iterative spring embedder is used. After use, it typically enables the spring embedder to complete a three-dimensional graph drawing with fewer iterations. The first phase of the preprocessor obtains an initial layout where all edge lengths are close to the user's predefined ideal length. This is achieved by means of an iterative procedure where nodes are moved to a position that is based on the location of its neighbours and the direction of the emanating edges. This process is repeated a number of times. It is much faster than the spring embedder, as each iteration of the preprocessor has linear time complexity, O(|E|), where |E| is the number of edges (hyperlinks) in the graph. The classical spring embedder has $O(|N|^2)$ complexity for each iteration it performs, where |N| is the number of nodes (pages) in the graph.

The second phase simply attempts to ensure an even distribution of nodes by placing the nodes on a uniform grid. The first phase often leaves clusters of nodes very close together, however a simple conflict resolution method is used to place nodes at the next available location, ensuring that nodes remain reasonably close to their original position when allocated to the grid. This is much quicker than the first phase, as it is not an iterative process. Typically we expect the performance of this phase to be close to linear, as we have noticed that most WWW graphs result in a low occurrence of nodes occupying the same grid location.

Section 2 describes our method in more detail. Section 3 gives comparative results of using our method against using spring embedding without preprocessing and also shows the visualization of WWW sites with our method. Section 4 gives our conclusions and proposes some further research.

2. Description of the Preprocessor

For our experiments, we consider connected WWW graphs in three-dimensional space, with each page being represented by a node and hyperlinks being represented by edges that connect pairs of nodes. We define a graph G = (N,E), where N is the set of nodes and E is the set of edges that connect nodes together.

Our preprocessor is a two phase procedure. The first phase attempts to obtain an initial drawing where all edge lengths are approximately the same. We refer to this ideal edge length as ka, where k is a constant representing the desired minimum node separation in the final graph drawing and $a \geq 1$. The motivation for using edge lengths larger than k for phase 1 is that we have observed that this improves the quality and reduces the number of spring embedder iterations required to produce a good final drawing. Preliminary experimentation suggests that ideal value of a is dependant on the mean number of emanating edges. The first stage of our preprocessor is to allocate the nodes of the graph over a very large volume using a uniform random distribution. Our experiments suggest that the preprocessor works more effectively when rapidly shrinking large random graphs. These are restricted to within a cubic volume of side length $10^3 ka(|N|^{1/3})$, which is normally sufficient to ensure good results by allowing the graph layout to contract.

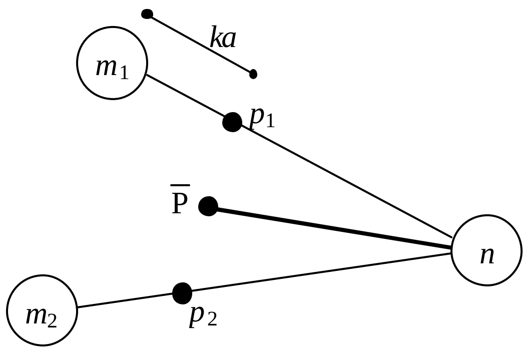

Figure 1

Each iteration of our preprocessor involves visiting every node of the graph in an arbitrary order. For each node n that we visit, we examine the set of emanating edges, F, each of which connects n to m_i. For each edge, we calculate the position of point p_i such that the vector $m_i p_i = ka(u_i)$, where u_i is the unit vector of $m_i n$. We then move node n to the mean position of p_i before repeating this process with the next node in the graph [Figure 1]. After a number of iterations, the graph is shrunk into a more stable graph where each edge length is near to ka.

The second phase is used to achieve the minimum node separation. We impose the constraint that all nodes must be allocated to a unique location on a three-dimensional grid of unit size k (the minimum node separation value). We perform this task by assigning each node to its nearest free location on the k-grid. If a conflict occurs, because the desired location has already been occupied by another node, then we pick the next nearest location. A variety of strategies exist for this conflict resolution. In our implementation, we have chosen to search through the surface of a cube centred about the desired (but occupied) location. This cube continues to grow larger until a free location is found and the node is allocated to it. After we have allocated every node, we end up with a new graph layout where edge lengths remain close to the phase 1 ideal value of ka, whilst ensuring all nodes have the final graph ideal minimum node separation of k. This is the *initial drawing* to which we apply the spring embedder.

To summarise in pseudocode: -

```
Phase1:
FOREACH iteration
 FOREACH node IN graph
  F = setOfEmanatingEdges(node)
  P = {}
  FOREACH edge IN F
   Calculate location of $p_i$
   Add $p_i$ to P
  Move node to mean location in P

Phase2:
FOREACH node IN graph
 Move node to nearest free $k$-grid slot
```

We follow these two phases with the application of a simple spring embedder based on that of Fruchterman and Reingold [4]. This version of the spring embedder is effective and widely used. For our purposes it also has the advantage of using k, the ideal minimum node separation in both attractive and repulsive forces. In this model, the repulsive forces between nodes are $-k^2/d$ and the attractive forces due to edges are d^2/k, where d is the distance between the two nodes.

It is worth noting that the second phase of the preprocessor may also be used after spring embedding in order to guarantee a minimum node separation of k, enabling three-dimensional node labels of this size to be accommodated in the final drawing. It could be argued that a number of iterations of the spring embedder, neglecting all non-edge forces, could mimic the behaviour of the first phase. This could take a long time to settle if very small movements are made by nodes on each iteration. Conversely, if the movements are too large then oscillations and overshoots are possible. The first stage also offers an advantage over the "edge force only" spring embedder in that it is independent of any parameters that determine how far nodes travel per iteration.

3. WWW Visualization results

To determine the benefit of using the preprocessor, we have compared the use of our preprocessor followed by spring embedding against using spring embedding alone. In our comparisons, we consider the spring embedder to have finished when we perform an iteration where all nodes move less than $k/100$, where k is the ideal edge length, as defined in Section 2.

The spring embedding alone has been applied to an initial graph where all nodes have been allocated to a random position in three-dimensional space. The initial random graph occupies a similar volume to the final drawn graph by constraining nodes to within a cubic volume of side length $k(|N|^{1/3})$. We believe this to be a fair starting point as experimentation suggests that there is no discernible advantage to be achieved by varying this initial volume by a relatively small amount, although when the volume is very large or small, a disadvantage becomes apparent.

An experimentation framework, containing the preprocessor and spring embedder, was implemented in Java. All experiments were carried out with Sun's Java 1.4 Virtual Machine on a Pentium 4 1.5GHz desktop machine with 512mb RAM.

Our experimentation framework was used to produce drawings of ten WWW sites, varying in size from 18 to 463 pages. We specified the following constant values: $k = 10$, $a = 5$. We began our first set of drawings by using both phases of the preprocessor, followed by the spring embedder. Our second set of drawings was obtained by applying only the spring embedder.

Table 1

Graph	Site	|N|	|E|
A	www.web-bits.net	18	49
B	www.brettmeyers.com	23	58
C	www.a-spotted-dog.com	31	127
D	www.jibble.org	72	417
E	www.bersirc.com	104	801
F	www.ivarjohnson.com	111	223
G	www.aspmessageboard.com	172	2050
H	www.peacenikjive.com	226	667
I	www.xml101.com	321	4327
J	www.i-scream.org.uk	463	4585

The source data is given in Table 1. It was obtained between 11 Feb and 26 Feb 2002. The authors have been involved in the development of two of these sites: www.jibble.org and www.i-scream.org.uk. The remaining sites were randomly selected from a Google Web Directory listing of free web design and development sites.

Table 2

	With Preprocessor		Without Preprocessor	
Graph	I_p	I_s	I_p	I_s
A	200	112	0	48
B	200	190	0	113
C	200	198	0	261
D	200	195	0	600
E	200	140	0	992
F	200	201	0	1700
G	200	172	0	1958
H	200	263	0	3396
I	200	268	0	3925
J	200	332	0	4263

Each drawing was created ten times, each using a different initial random layout. We recorded the number of iterations required for the graph drawing to reach an equilibrium. The mean values of these results for each graph are shown in Table 2. I_p shows the number of iterations of preprocessing and I_s shows how many spring embedder iterations were required to finish the drawing. When drawing all the graphs, 200 preprocessor iterations were applied. We have yet to determine the ideal number of pre-processor iterations for a given graph, indeed, it has been observed that better results are often obtained if it is not run fully to an equilibrium.

We can see from the above results that our two smallest graphs did not benefit from the effect of the preprocessor. Both of these required more iterations of the spring embedder to finish drawing the graph when the preprocessor was used. However, as the preprocessed graphs get larger, we observe significant reductions in the number of spring embedder iterations required to bring the layout to an equilibrium configuration.

The time complexity of a single preprocessor iteration is linear with respect to the number of edges in the graph. As the size of each graph increases, the time taken to execute a preprocessor iteration becomes much less than the time taken to execute a single iteration of the spring embedder. We can therefore justify the expense of applying many iterations of the preprocessor on larger graphs, as the time spent doing so will be negligible compared with the time saved by reducing the number of spring embedder iterations required to complete the drawing.

Table 3

	With Preprocessor			Without Preprocessor
Graph	**T_p**	**T_s**	**T_p+T_s**	**T_s**
A	0.009	0.030	0.039	0.012
B	0.011	0.069	0.080	0.041
C	0.021	0.116	0.137	0.154
D	0.068	0.463	0.531	1.424
E	0.118	0.653	0.771	4.630
F	0.048	0.988	1.036	8.359
G	0.330	2.267	2.597	25.81
H	0.122	5.126	5.248	66.18
I	0.885	8.919	9.804	130.6
J	0.987	21.85	22.84	280.6

Table 3 shows the actual average time in seconds taken to produce each drawing using our experimentation framework. T_p is the amount of time spent applying the preprocessor and T_s is the time taken to complete the graph drawing using the spring embedder.

The improvement in performance as the graphs get larger is reflected in these timing results, where the preprocessing time becomes less significant as the size of graph increases and so the benefit from reducing the number of spring embedder iterations is clear. An overall speed up of more than 10 times is apparent in the largest four graphs. As the software is implemented with little emphasis on time performance, we believe better optimised code will produce even faster results.

We illustrate some of these initial and final graph drawings in Figures 2-9. In Figures 10-11 we show the effect of the preprocessor without subsequent spring embedding on some graphs, but it should be noted that most graphs are not amenable to preprocessing only.

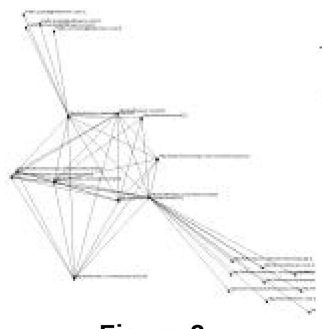

Figure 2

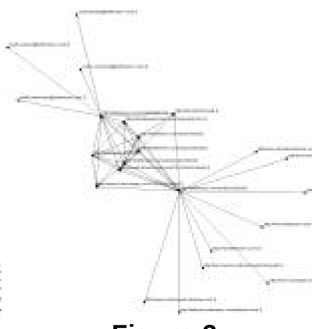

Figure 3

Figure 2 is the result of applying 200 iterations of preprocessing to Graph B (23 nodes). This graph was drawn from a large random layout, which rapidly shrinks to form this stable initial drawing. Each edge length is approximately ka and all nodes are separated in three-dimensional space by at least k. This graph is shown labelled with the pages in the site, but this labelling is not practical for larger graphs. Figure 3 shows Graph B after spring embedding the preprocessed graph [Figure 2] to equilibrium. On this small graph, we see no advantage in using the preprocessor. It is, in fact, quicker to draw this graph without using the preprocessor, unlike the larger graphs later in this paper.

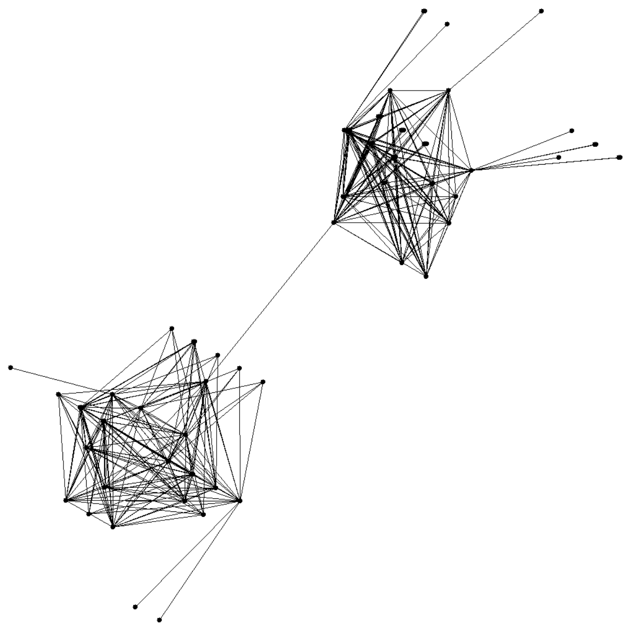

Figure 4
Graph D (72 nodes) after 200 iterations of preprocessing. We can see that the preprocessor has produced an initial drawing consisting of two clusters. Notice that there is only a single hyperlink connecting the two clusters, indicating that navigation of the web site could be improved.

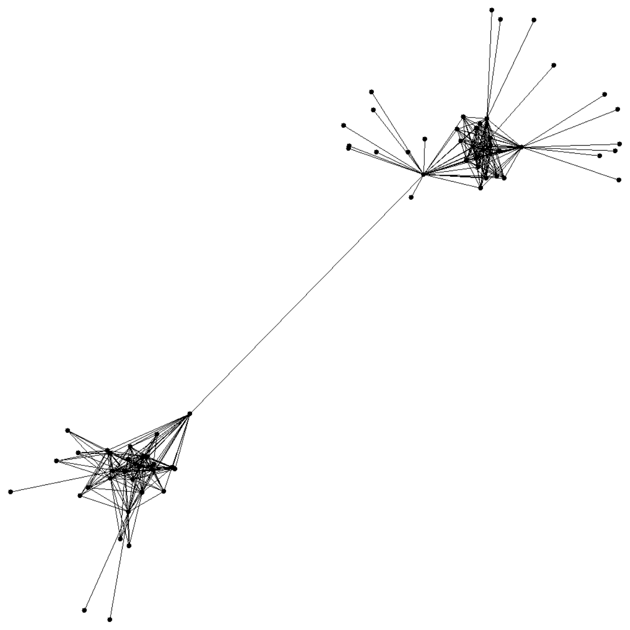

Figure 5
Graph D after spring embedding the preprocessed graph to equilibrium. We can see that the final graph holds a similar shape to the preprocessed graph [Figure 4]. The minimum node separation of k imposed by the preprocessor is useful for preventing large forces between pairs of nodes which may cause an initial disruption to the drawing by creating a large node displacement.

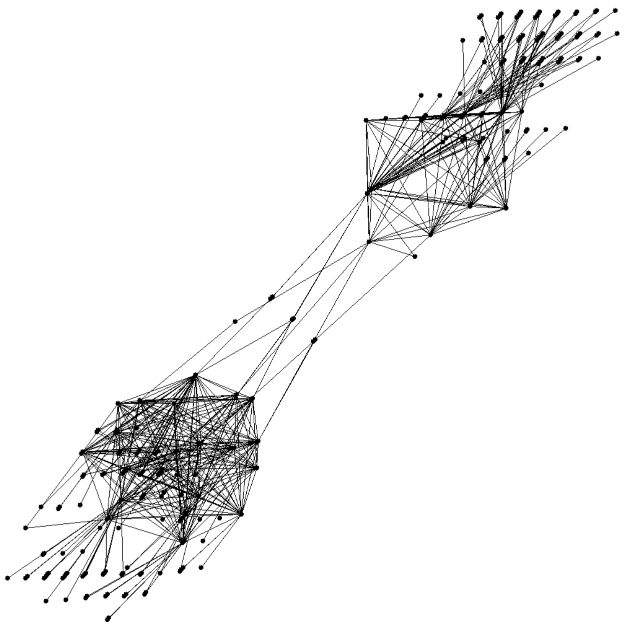

Figure 6

Graph H (226 nodes) after 200 iterations of preprocessing. In this example, we see more clearly the allocation of nodes to the k-grid. The use of the k-grid is very beneficial in this case, as it has prevented many nodes from occupying the same small region of three-dimensional space. Several different paths link the two clusters that have become apparent in this initial drawing.

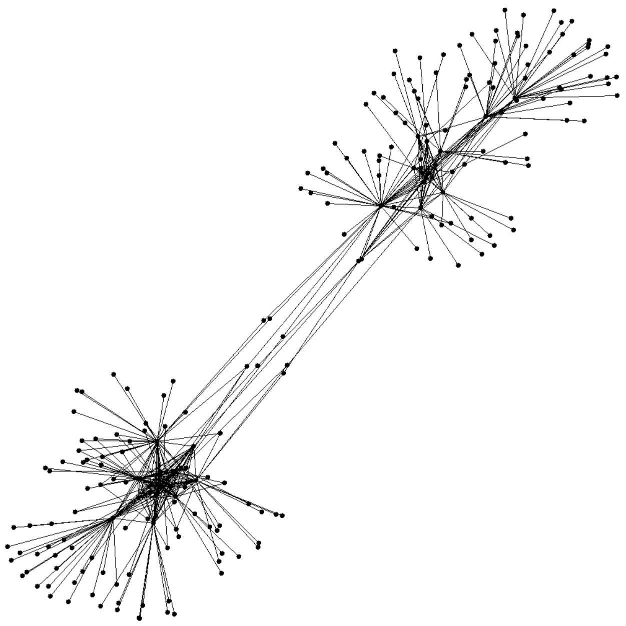

Figure 7

Graph H after spring embedding the preprocessed graph [Figure 6] to equilibrium. The general shape of the graph has not changed a great deal and we can still see the same paths between the two clusters. Once again, running the spring embedder on the preprocessed graph offers a reduced number of iterations to complete the drawing.

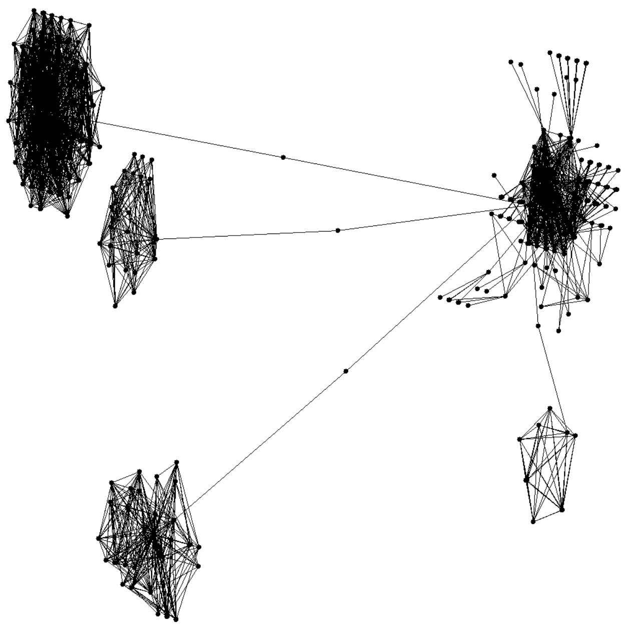

Figure 8

Graph J (463 nodes) after 200 iterations of preprocessing. The preprocessor has rapidly shrunk the large random graph to produce this drawing with 5 apparent clusters. Assignment of nodes to the k-grid has been carried out in the second phase of the preprocessor, resolving the issue of nodes that end up separated by a small Euclidian distance.

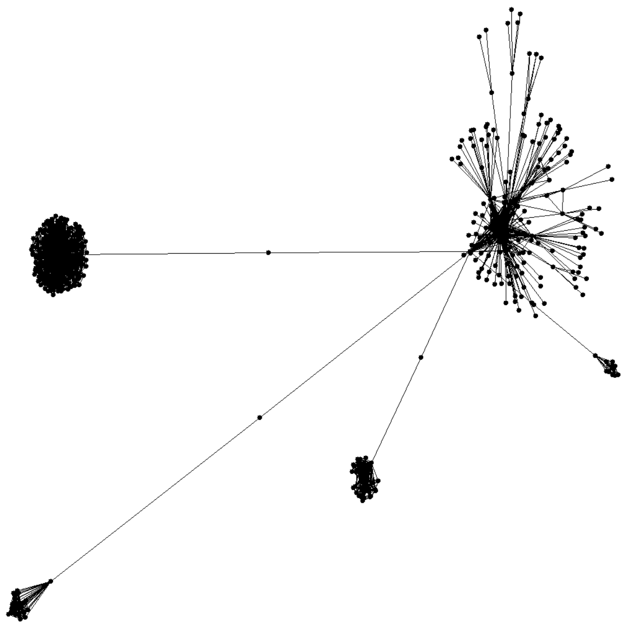

Figure 9

Graph J after spring embedding the preprocessed graph [Figure 8] to equilibrium. Note that we can still see the 5 clusters previously identified by the preprocessor. Using the preprocessor here has clearly been advantageous, as we have been able to draw this in under 23 seconds (including time taken to apply the preprocessor), compared with the 280 seconds required to draw the graph from a random layout with the spring embedder alone.

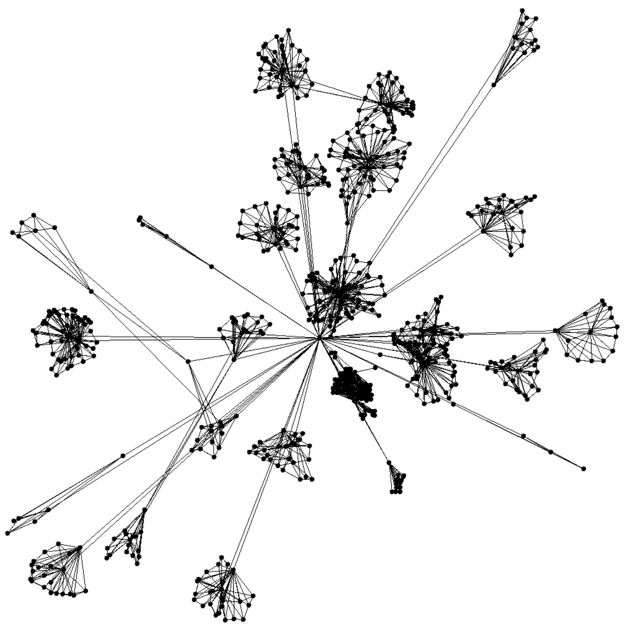

Figure 10

A WWW graph (http://spod.cx) consisting of 700 pages and 3255 hyperlinks. This drawing was produced in 0.025 seconds by applying 5 iterations of the preprocessor. This is one of a very few cases where the preprocessor can produce a good drawing in a very short amount of time without requiring subsequent spring embedding.

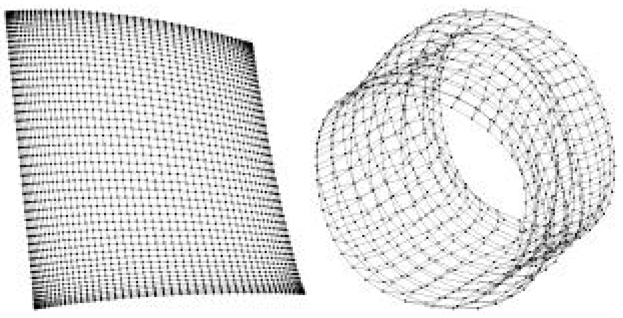

Figure 11

The 2500-node grid was drawn in under 3 seconds using 400 iterations of the first stage of preprocessing and no k-grid allocation or subsequent spring embedding. The ring graph took less than a second using 300 iterations. The fast generation of these types of graphs shows that the preprocessor may have possible application areas other than being used for WWW visualization.

4. Conclusions

We have shown how a novel preprocessor can produce an initial three-dimensional layout of a WWW site graph that reduces the number of spring embedder iterations required to produce a good drawing.

In some cases, the preprocessor alone can produce adequate graph drawings. In many graphs, clusters can be displayed without requiring subsequent spring embedding. We have also observed that clusters are best displayed if the preprocessor is not run to equilibrium, as the spring embedder appears to achieve better results in these cases.

These are preliminary results and so improved performance may be achieved. Needing further investigation is the ideal number of preprocessor iterations for a given graph in order to produce better drawings of links between individual clusters. The performance of the preprocessor could be improved by including numerical optimisations such as the conjugate gradient method used by JIGGLE [12]. The method can also be made more attractive by using an improved spring embedding method in the final stages, such as the FADE algorithm [11].

Investigation is still required into the sizes and types of graph for which our preprocessor graph drawing method is a useful tool. We have not yet demonstrated the benefit of using the preprocessor for drawing very large graphs, nor have we attempted much experimentation in other graph drawing application areas.

References

1 I. Bruβ, A. Frick. Fast Interactive 3-D Graph Visualization. LNCS 1027. pp. 99-110. 1996.

2 P. Eades. A Heuristic for Graph Drawing. Congressus Numerantium 42. pp. 149-60. 1984.

3 A. Frick, A. Ludwig, H. Mehldau. A Fast Adaptive Layout Algorithm for Undirected Graphs. GD'94, LNCS 894. pp. 388-403. 1995.

4 T. M. J. Fruchterman, E. M. Reingold. Graph Drawing by Force-directed Placement. Software – Practice and Experience Vol 21(11). pp. 1129-1164. 1991.

5 D. Harel, Y. Koren. A Fast Multi-scale Method for Drawing Large Graphs. GD 2000, LNCS 1984. pp. 183-196. 2001.

6 M.L. Huang, P. Eades, R. F. Cohen. WebOFDAV — navigating and visualizing the Web on-line with animated context swapping. WWW7: 7th International World Wide Web Conference. 1998.

7 D.S. McCrickard & C.M. Kehoe. Visualizing Search Results using SQWID. WWW6: 6th International World Wide Web Conference. 1997.

8 T. Munzner and P. Burchard. Visualizing the Structure of the World Wide Web in 3D Hyperbolic Space. VRML '95, special issue of Computer Graphics, ACM SIGGRAPH, pp. 33-38. 1995.

9 G. Parker, G. Franck, C. Ware. Visualization of Large Nested Graphs in 3D: Navigation and Interaction. J. Visual Languages and Computing, 9(3). pp. 299-317. 1998.

10 A. Quigley. Large Scale Relational Information Visualization, Clustering and Abstraction. Thesis, University of Newcastle, Australia. 2001.

11 A. Quigley, P. Eades. FADE: Graph Drawing, Clustering, and Visual Abstraction. GD 2000, LNCS 1984. pp. 197-210. 2001.

12 D. Tunkelang. JIGGLE: Java Interactive Graph Layout Algorithm. GD '98, LNCS 1547. pp. 413-422. 1998.

13 C. Walshaw. A Multilevel Algorithm for Force-Directed Graph Drawing. GD 2000, LNCS 1984. pp. 171-182. 2001.

14 G.J. Wills. NicheWorks — Interactive Visualization of Very Large Graphs. J. Computational and Graphical Statistics 8,2 pp. 190-212. 1999.