Some Three-Dimensional Graph Drawing Algorithms

A thesis submitted for the degree of
Master of Computer Science

by

Diethelm Ironi Ostry BSc(Hons)

Department of Computer Science
and Software Engineering

The University of Newcastle

October, 1996

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution.

Acknowledgements:

My thanks

to my supervisor Professor Peter Eades, for his invaluable and generous help in the form of insight, encouragement, inspiration and friendship,

to Dr. Robert Cohen for his technical help and fine humour,

to my friends, fellow students and the staff of the Department of Computer Science and Software Engineering at the University of Newcastle for their warm welcome to a uniquely stimulating environment,

to IBM (and especially to Dr. Arthur Ryman at the IBM Toronto Laboratory) for their support and a memorable introduction to Toronto,

to Dr. John Knight and Dr. René Grognard for helpful discussions,

and to Dr. John O'Sullivan and CSIRO Division of Radiophysics for their encouragement and support for this work.

To my mother and father

Contents

Abstract

In order to realise the potential benefits of three–dimensional (3D) display of relational information, there is a need for effective 3D human–computer interface designs. Algorithms for automatically creating 3D visual representations of relational information are a significant component of these interfaces.

One productive strategy for developing such algorithms has been via the *graph* as an intermediate representation of the relational information: the information is first expressed as a graph and then a *layout* algorithm is used to create a visual representation of the graph.

This thesis examines some technical issues which arise when several common layout algorithms, developed originally for 2D display of graphs, are extended specifically to 3D display.

Typical computer graphics display systems can only provide a limited resolution and display area. This places a limit on the size of graph which can be displayed effectively. Simplification of the graph can permit the display of larger graphs, or a produce a clearer display. One such simplification technique, *clique partitioning*, is examined, and a new heuristic algorithm for implementing it is proposed.

The force–directed approach to graph layout has proved to be a popular and effective technique for the generation of visual representations of undirected graphs. A general differential equation formulation of this approach is given. It is shown that a large class of drawing constraints can be readily incorporated in this formulation. A numerical solution of the underlying system of equations using standard techniques can be excessively slow due to a property known as *stiffness*. Examples are given showing the considerable improvement in speed when special–purpose numerical solution techniques are used.

When the graph representing the relational information is *directed*, the edge orientation carries significant information. Hierarchical layout algorithms attempt to display this information clearly. This approach is extended to the display of directed graphs with their vertices confined to 3D surfaces.

Chapter 1

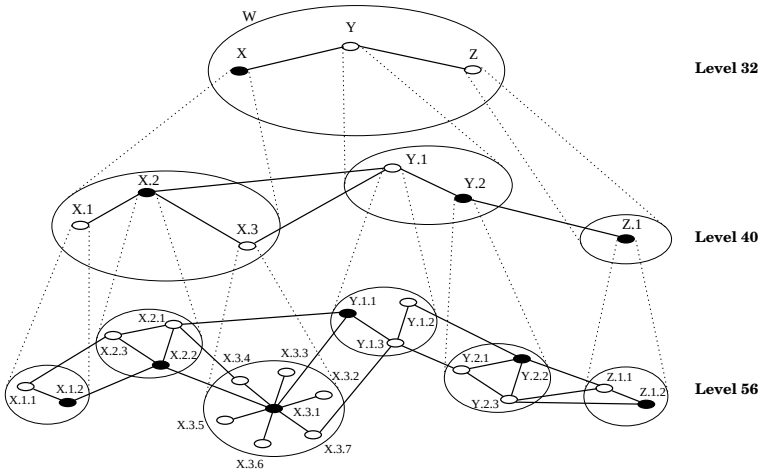
Introduction

> We should talk less and draw more. I personally would like to renounce speech altogether and, like organic nature, communicate everything I have to say in sketches.
>
> Goethe

1.1 Visualisation of Relational Information

It seems almost trite to comment on the extraordinary power of images to convey complex information. Certainly the field of scientific data visualisation has undergone a revolution in the past three decades through the adoption of computer–generated pictorial displays of numerical information (see for example [1, 2]). The human perception system is much better adapted to interpreting drawings in two and three dimensions than simple tables of numbers. The use of automatically–generated pictorial displays to help interpret *relational* information is not yet widespread, although a large body of techniques for displaying relational information as diagrams has been developed in the same three decades. Diagrammatic user interfaces have been developed in the areas of visual programming and CASE tools, database schema representation, and for other CAD systems (see for example [3, 4, 5, 6, 7, 8]). Figure 1.1 illustrates some advantages of a graphical representation of complex relational information.

The task of visualising relational information is to produce an easily–interpretable



Level 32

Peer Group(W):
{X,Y,Z}
{(X,Y), (Y,Z)}

Level 40

Peer Group(X):
{X.1, X.2, X.3}
{(X.1, X.2), (X.2, X.3)}

Peer Group(Y):
{(Y.1, Y.2)}

Peer Group(Z):
{Z.1}

{(X.2, Y.1), (X.3, Y.1), (Y.2, Z.1)}

Level 56

Peer Group(X.1):
{X.1.1, X.1.2}
{(X.1.1, X.1.2)}

Peer Group(X.2):
{X.2.1, X.2.2, X.2.3}
{(X.2.1, X.2.2), (x.2.2, X.2.3), (X.2.3, X.2.1)}

Peer Group(X.3):
{X.3.1, X.3.2, X.3.3, X.3.4, X.3.5, X.3.6, X.3.7}
{(X.3.1, X.3.2), (X.3.1, X.3.3), (X.3.1, X.3.4),
X.3.1, X.3.5), (X.3.1, X.3.6), (X.3.1, X.3.7)}

Peer Group(Y.1):
{Y.1.1, Y.1.2, Y.1.3}
{(Y.1.1, Y.1.3), (Y.1.2, Y.1.3)}

Peer Group(Y.2):
{Y.2.1, Y.2.2, Y.2.3}
{(Y.2.1, Y.2.2), (Y.2.2, Y.2.3), (Y.2.3, Y.2.1)}

Peer Group(Z.1):
{Z.1..1, Z.1.2}
{(Z.1.1, Z.1.2)}

{(X.1.1, X.2.3), (X.1.2, X.2.3), (X.2.1, X.3.4), (X.2.2, X.3.1),
(X.2.1, Y.1.1), (X.3.1, Y.1.1), (X.3.7, Y.1.3), (Y.1.2, Y.2.2),
(Y.1.3, Y.2.1), (Y.2.2, Z.1.1), (Y.2.3, Z.1.1), (Y.2.3, Z.1.2)}

Figure 1.1: A routeing hierarchy for a broadband telecommunications network shown both schematically and as a table (adapted from The ATM Forum Private Network-Network Interface Specification). The diagram illustrates the use of *graphs* to display relational information. Communications links are represented by solid lines and network nodes are represented by graph vertices. Aggregation of vertices and edges in a recursive hierarchy provides a scalable abstraction for visualising the routeing information.

picture which conveys the application information. The most successful approach to automating this task is based on making an intermediate model of the information to be displayed. A mathematical model of the information is extracted from the application, and then a drawing is made of the model. This approach has been called "the visualisation pipeline" by some authors (see for example [9]). Figure 1.2 shows this approach. The modeling step extracts the information to be visualised and places it in the intermediate model form, the layout step assigns geometric attributes (such as coordinates) to the model components, and the rendering step generates a picture representing the model.

The most common model for relational information is the *graph*. A graph is a mathematical object comprising a set of *vertices* or *nodes* and a set of *edges* linking some of the vertices. The graph may be *directed*, when a particular direction is associated with an edge, or *undirected*. As a model for relational information, the graph is sufficiently complex to be a comprehensive representation for many applications, and simple enough to have stimulated a large body of theoretical work directed towards elucidating useful properties and algorithms.

The central problem of this approach to automatic visualisation of relational information thus becomes creating a drawing of a graph, or *graph layout*.

1.2 User Requirements

An abstract graph is described simply by the set of its vertices and the set of its edges. No particular geometry is associated with it. The graph layout operation embeds the abstract graph in some geometric space, i.e. it assigns a position to each vertex and a curve to each edge. The aim is to find a geometrical configuration of the vertices and edges such that the arrangement is easy to understand, and fits into the viewing area of the display device. This task can be characterised by a list of *user requirements* for a visualisation [10].

A summary of the user requirements for graph layout includes:

Readability The information contained in the graph should be easy to *read*, that is, it should effectively display the information of interest to the user. Most drawing techniques have approached this requirement by simultaneously enforcing *readability criteria* such as

Figure 1.2: The Information Visualisation Pipeline.

- the drawing should minimise the number of edge crossings,
- the drawing should maximise the angle between adjacent or crossing edges,
- the drawing should exhibit structural properties of the graph which are of particular interest to the user (for example, symmetries in the graph),
- vertices should be distributed uniformly in the viewing area with adequate *vertex resolution*, defined as the minimum distance between vertices in the drawing. Pairs of adjacent vertices should have similar separations, and unadjacent vertices should not be drawn close to each other.

Conformance Different application areas have historically developed different conventions for diagrammatically expressing the syntax and semantics of the information. The example in Figure 1.1 shows a *hierarchical* organisation, the use of circles to represent significant entities and groupings, and dashed and solid lines to show relationships. The *conformance criteria* are the rules defining a drawing style which conforms to the conventions of the application area.

Adaptability A user may wish to control some aspects of the drawing. For example, a user may prefer to place a particular graphical object at the centre of the drawing region, and limit the region to some specified size.

Efficiency The drawing algorithms should complete the display in a time acceptable to the user, typically within two seconds for interactive use [11].

The most successful approach to developing algorithms satisfying the readability requirements uses the idea of *drawing aesthetics*. An attempt is made to identify those properties (aesthetics) which make a particular drawing style successful in representing specific classes of graphs (e.g. trees) in an easily understood way. A simple example of a drawing aesthetic might be "edges should be drawn as straight lines of the same length". A drawing is accepted if it satisfies, or maximises compliance with, the chosen aesthetics.

The choice of a set of drawing aesthetics is crucial. If the constraints imposed by the selected aesthetics are too loose, there may be no unique drawing, while

if they are too severe, the situation might arise that no drawing satisfies all the aesthetics simultaneously. In the former case, additional constraints are needed to uniquely determine a drawing. In the latter, the aesthetics conflict and a compromise between them is required so that the drawing approximately satisfies all the aesthetics. An optimum choice (in this sense) prescribes a set of aesthetics, and assigns a relative importance to each, so that a unique drawing results.

1.3 2D Graph Layout

Most research in graph layout to date has been directed towards the problem of drawing graphs in two dimensions. An indication of the interest in this problem during the last decade is the size of the annotated bibliography given in [12]. Conferences on this subject are held regularly [13].

The most commonly implemented 2D layout algorithms are based on the *topology-shape* approach, the *hierarchical* approach and the *force–directed* approach [14]. In the first two approaches, potential conflicts between drawing aesthetics are handled by decomposing the drawing procedure into a sequence of steps. Each step produces a refinement of the drawing while satisfying one or more of the drawing aesthetics. In the force–directed approach, a relative importance is attached to each aesthetic constraint and a drawing approximately satisfying all the constraints is sought.

The topology-shape approach [15, 16, 17] constructs orthogonal drawings. In an orthogonal drawing, the edges of the graph are represented by piecewise-straight lines whose segments are orthogonal (see Figure 1.3(f) for an example). Topology and shape are properties of orthogonal drawings which establish equivalence classes in orthogonal drawings of the same graph:

- two orthogonal drawings have the same topology if one can be derived from the other by a continuous deformation which does not alter the sequence of edges around closed loops in the drawing, and
- two orthogonal drawings have the same shape if they have the same topology, and are related by changes in edge lengths.

The general procedure taken in this approach is to satisfy the user requirements as

nearly as possible while consecutively establishing the topology and shape properties of the drawing.

The hierarchical approach [18, 19] is particularly suited to directed graphs. The vertices of an acyclic directed graph can always be distributed among a sequence of ordered layers so that all edges can be drawn with a downward (or upward) component. This arrangement generally gives a much clearer display of the relational information represented by the graph than an arrangement with arbitrary edge orientations. The general procedure is a sequence of steps, each of which enforces a particular drawing aesthetic. The graph is temporarily made acyclic if necessary, the vertices are assigned to a set of ordered layers, and then arranged within each layer so that the number of edge crossings is minimised. At each step the user requirements provide additional constraints on the drawing. The hierarchical approach will be described in more detail in Chapter 4.

Finally, force–based approaches generate a mathematical description of a physical model of the graph. The user requirements are expressed as properties of, and constraints on the model so that at equilibrium the model approximately satisfies the readability requirements. The spring algorithm [20] is an example of this approach and will be discussed in more detail in Chapter 3.

A number of commercial and research systems for automatically generating 2D graph layouts based on these approaches have been developed (see for example [21, 22, 23, 24, 25]).

1.4 3D Graph Layout

Although most attention has focused on 2D graph layout, there is good reason to expect that making use of three-dimensional display can offer benefits in visualising relational information [26, 27, 28, 29]. The extra freedom in placing vertices may allow a more natural representation of the information. Figure 1.3 shows the same graph drawn in several different ways. Figure 1.3(d), intended to represent the graph in three dimensions, seems to best express the property that all the vertices have equivalent edge relations with their neighbours (Figure 1.3(a) could also be interpreted in this way, as a cube seen from a particular perspective).

There is a growing number of experimental studies [30, 31, 32] of perception of

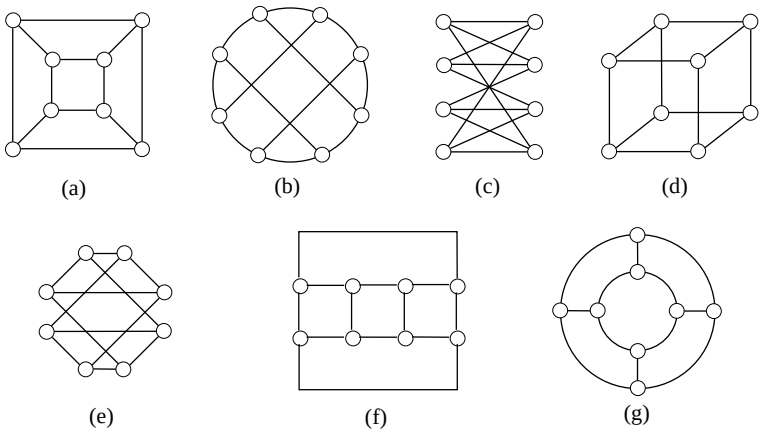


Figure 1.3: Different drawings of the same graph.

graphs drawn in 3D. Some of this work suggests that the error rate in identifying connectivity between vertices using a 3D picture of a random graph is roughly linearly proportional to the number of vertices, and that the error rate is reduced by about three times by displaying the graph in stereoscopic 3D. Equivalently, graphs with three times the number of vertices can be displayed at about the same error rate by using 3D.

Viewpoint rotations form a natural and strongly intuitive set of transformations for understanding a structure in 3D, akin to turning an unfamiliar object over in one's hands. This can viewed as a simple way of implementing an adaptability requirement for user–controlled placement of a designated vertex.

Finally, display devices and hardware supporting 3D graphical displays are becoming more common and show a similar exponential fall in price/performance ratio over time to that of workstations.

Some simple 3D extensions of the hierarchical and force–directed approaches to graph layout will be described in Chapters 3 and 4. One motivation for this work is to demonstrate a specifically three–dimensional drawing aesthetic: *vertices should be placed on simple surfaces.*

1.5 Edge Density

A distinction can be made between *dense* graphs, that is, graphs which have a large number of edges relative to the number of vertices, and *sparse* graphs. A

high density tends to make for confusing pictures. Edge crossings are likely to be unavoidable in the drawing, or so many edges are displayed that it becomes difficult to associate their endpoints visually.

One general strategy for reducing this problem is to pre–process the graph in some way so that the number of edges to be displayed is reduced. An example which will be discussed in the next chapter is to partition the vertices of the graph into subgraphs (cliques) and, in a hierarchical fashion, represent each subgraph as a single vertex ("vertex concentration"). The simplified graph has fewer vertices and edges, making techniques for sparse graph layout more effective.

1.6 Contributions of the Thesis

The main concern of this thesis is the extension to 3D of some common graph layout algorithms, while addressing user requirements.

In Chapter 2, a method for simplifying dense graphs is presented. A heuristic method for clique partition is developed to maximise the number of edges included in the cliques. Experimental analysis shows that the proposed method combines the useful properties of some other clique–partition algorithms in this application.

In Chapter 3, an elaboration of the differential equation formulation of the spring algorithm [20] is presented. This formulation is shown to allow simple inclusion of surface constraints to satisfy a range of readability, conformance and adaptability requirements within a unified framework. Numerical solution of the differential equations is discussed, and it is shown that these equations can display *stiffness*, which requires special–purpose numerical algorithms for the greatest efficiency (i.e. execution speed).

In Chapter 4, several 3D extensions of the hierarchical approach [18] to the layout problem for directed graphs are presented. Included is a hybrid approach which combines hierarchical layering with a 3D extension of force–directed methods.

Chapter 2

Vertex Concentration

2.1 Partitioning for Graph Simplification

Graph layout algorithms are generally targeted at specific, limited classes of graphs. Their principal aim is to make the relational information represented by the graph understandable. In this, they may perform poorly for graphs outside their main range of application. Many drawing algorithms work best for sparse graphs, that is, graphs with relatively few edges.

For dense graphs, that is, graphs which have a relatively large number of edges (the complete graphs K_N being the extreme examples), drawings tend to appear cluttered with intersecting edges. Three dimensional layout and viewing of these graphs can reduce the problem to some extent. For hierarchical drawings of directed graphs, an *edge concentrator* [33] approach has been proposed in which multiple crossing edges between adjacent layers are represented by a single graphical object. There has also been interest in a pre–processing step in which the vertices of the original graph is partitioned into disjoint subgraphs with specific properties.

This suggests a hierarchical approach to visualising the graph: at the top level, the subgraphs and parallel edges between them are represented as simplified icons, reducing the number of edges and vertices to be displayed. A reduction in the time required to generate a layout of the simplified graph can be expected. At lower levels, subgraphs and multiple edges can be expanded under user control using a

drawing style which makes use of the special properties of the subgraphs.

The partitioning can be done on a semantic or graph–theoretic basis. Semantic partitioning groups vertices together into subgraphs on the basis of some common property in the application domain. In the simplest case, a user arbitrarily assigns selected vertices to subgraphs. The use of semantic partitioning naturally leads to a requirement for algorithms for manipulating and displaying compound graphs [34, 35].

Here we focus on vertex–partitioning approaches based on graph–theoretic properties. Techniques which appear to be applicable for this purpose partition the graph into planar subgraphs, *clans* [36] or cliques. In this chapter, clique partition will be investigated as a simplifying pre–processing step to improve the applicability of graph layout algorithms to dense graphs.

In a planar subgraph partition, each subgraph can be drawn in the plane without edge crossings. For a graph with a partition of just two planar subgraphs, the subgraphs can be drawn on parallel planes in 3D, with edges connecting the subgraphs lying between the planes. For partitions with more subgraphs, each can be drawn on one face of a convex polyhedron, with edges between subgraphs traversing its interior.

A clan–based graph decomposition [36] decomposes a directed acyclic graph into clans [37, 38]. Given a directed acyclic graph G, then a subset $X \subseteq G$ is a clan *iff* for all x, $y \in X$ and all $z \in G - X$, (a) z is an ancestor of x *iff* z is an ancestor of y, and (b) z is a descendant of x *iff* z is a descendant of y. The decomposition parses the directed graph into a tree of subgraphs which can be drawn by adapting hierarchical tree–drawing algorithms (see for example [36, 26]).

2.2 Vertex–disjoint Clique Partition

This method of graph simplification partitions the vertices of the graph into subgraphs, each of which is a clique (a *clique* is a subgraph which is complete, i.e. all vertex pairs in the subgraph are connected by an edge) (see Figure 2.1). Each vertex is contained in exactly one clique. This has also been called a *clique cover*[39]. Since the vertices within each clique are connected in a known *a priori* way, no information is lost by displaying the clique as a simple compound vertex or icon

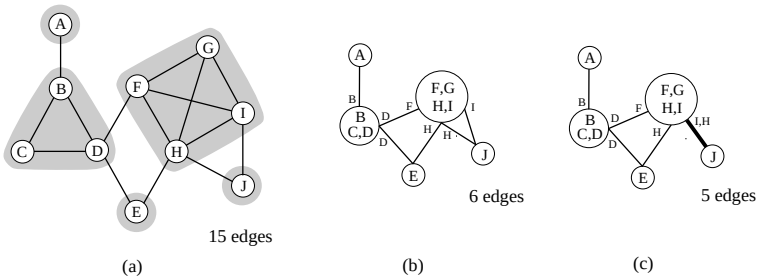


Figure 2.1: A vertex disjoint clique partition (a) forms the basis for a drawing (b) which reduces the number of edges to be displayed. Parallel edges can be drawn as single edges (c) with some loss of information in general.

with edges labeled to identify the (hidden) interior vertices to which they connect.

Dense graphs are likely to contain non–trivial cliques. The edge structure of a clique is implicit and so the clique can be drawn as a single icon. The edges contained within the clique do not need to be explicitly displayed, thus reducing the edge–density of the displayed graph. Large cliques of course will result in the greatest reduction of the number of edges to be displayed.

Bhasker *et al* [40] give an optimal upper bound θ_{upper} on the minimum number of cliques into which a graph can be partitioned. For a simple undirected graph G with n vertices and e edges,

$$\theta_{upper} = \left\lfloor \frac{1 + \sqrt{4n^2 - 4n - 8e + 1}}{2} \right\rfloor$$

Defining the density ρ as the ratio of the number e of edges in the graph to the number of edges in a complete graph with the same number n of vertices,

$$\rho = \frac{e}{\binom{n}{2}} = \frac{2e}{n^2 - n}$$

we have

$$\theta_{upper} = \left\lfloor \frac{1 + \sqrt{4n(n-1)(1-\rho) + 1}}{2} \right\rfloor \quad .$$

As ρ increases, θ_{upper} decreases. There must be more cliques than θ_{upper}, and so there must be a clique of size M with

$$M \geq \left\lfloor \frac{e}{\theta_{upper}} \right\rfloor \quad ,$$

which increases with ρ.

After carrying out a clique partition as a preprocessing stage, the simplified graph can be drawn using a suitable layout algorithm with the addition of edge labeling indicating to which vertex of a clique a particular external edge is connected. This expresses the same information as a drawing of the original graph. In general the final graph, comprising icons representing cliques and labeled edges between them, will not be simple, since multiple edges can occur between cliques. A hierarchical display could initially represent the graph as the cliques, represented as icons, connected by simple iconic edges representing the parallel edges. User interaction could control expansion of the icons as desired.

2.3 Maximum Included–Edge Clique Partition

The optimum clique partition minimises the number of edges to be displayed. This is equivalent to finding a clique partition which maximises the number of edges of the original graph which are contained within cliques.

A formal statement of this problem is:

Maximum Included Edge Clique Partition (MIECP)
Instance: An undirected simple graph G.
Task: Find a vertex–disjoint clique partition having the greatest number of edges of G included in cliques.

This variant of the clique partitioning problem does not seem to have been specifically considered in the literature, although good heuristics are known for finding clique partitions (for a review, see [40]).

2.3.1 Complexity

For practical application, it is of interest to establish the computational complexity of finding a clique partition which maximises the number of edges contained in cliques.

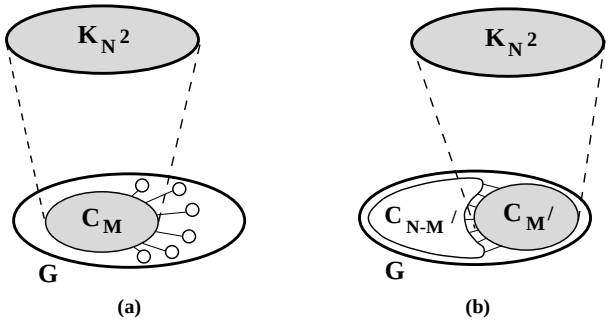


Figure 2.2: Extreme cases in establishing the complexity of the maximum included edge clique partition problem on a graph G with N vertices.

Theorem 2.1 (MIECP) *The maximum included–edge clique partition problem is NP–hard.*

Proof: (Reduction to an instance of Maximum Clique (**MC**, [41])). Let G be an undirected, simple, connected graph of N vertices. Now construct an augmented graph G' by appending the complete graph K_{N^2} to G with edges from each vertex of G to each of the N^2 vertices of K_{N^2}. The proof will be carried out by showing that a solution of **MIECP** on G' will combine a largest clique in G with the appended K_{N^2} to form the largest clique in the partition. **MIECP** on G' can thus be used to solve **MC** on G.

First, consider the number of edges contained in cliques in a clique partition where K_{N^2} and C_M, a clique of size M, combine to form a single clique and no other edges of G' are included in cliques (see Figure 2.2 (a)).

Let L_1 be the number of edges included in cliques for this case:

$$2L_1 = (N^2 + M)(N^2 + M - 1).$$

This is the minimum possible number of edges contained in the cliques of a clique partition of G' for any G containing a largest clique of size M.

Now consider the number of edges contained in cliques when K_{N^2} attaches to some clique of size M' in G, where $M' < M$ (see Figure 2.2 (b)). Let L_2 be the number of edges included in cliques for this case. Note that the largest possible value of L_2 occurs when G contains just one other clique of size $N - M'$ (since a clique of size n contains more edges than two cliques of size m_1 and m_2 where $n, m_1, m_2 > 0$ and $m_1 + m_2 = n$). In that case,

$$2L_2 = (N^2 + M')(N^2 + M' - 1) + (N - M')(N - M' - 1).$$

Let $k > 0$ be an integer such that $M = M' + k$. Then

$$2L_1 = (N^2 + M')(N^2 + M') + k(N^2 + M') + k(N^2 + M' + k - 1)$$

so

$$\begin{aligned} 2(L_1 - L_2) &= k(N^2 + M') + k(N^2 + M' + k - 1) - (N - M')(N - M' - 1) \\ &= 2k(N^2 + M') + k(k - 1) + M'(2N - M'). \end{aligned}$$

So $L_1 - L_2 > 0$ for all $k > 0$. A solution of **MIECP** on G' will thus attach K_{N^2} to a largest clique of G and consequently solve **MC** on G.

□

2.3.2 Heuristics

Efficient heuristic algorithms are known for the closely related *clique partition* problem for graphs, that is, the problem of finding the smallest number of cliques in a graph such that each vertex is contained within exactly one clique [40, 42]. These algorithms do not however directly attempt to maximise the number of edges contained within the cliques of the decomposition and may not yield a maximum included–edge clique partition. Figure 2.3 shows a new heuristic algorithm which attempts to generate a maximum included–edge clique partition. Experimental tests of this algorithm indicate that it generally combines the best features of both the Bhasker–Samad [40] and Tseng [42] algorithms when applied to the MIECP problem. .

The algorithm operates on an undirected graph and reduces it to a number of isolated cliques. In the representation acted on by the algorithm, vertices are taken to represent cliques.

The algorithm proceeds by selecting two adjacent vertices to combine into a new vertex. The selection is made on the basis of minimising the number of edges between the current cliques. Lists are maintained of the contents of each clique and the deleted edges between cliques.

1. For each edge of the graph, compute n_d, the number of edges which would be deleted in step 5 if this edge were selected. If the graph has no edges, STOP.

2. Choose the edge e with minimum n_d. If there is more than one edge with the minimum n_d, select the edge with most common neighbours. (A common neighbour of an edge is a vertex which has an edge to both end vertices of the edge.) If more than one candidate edge has the largest number of common neighbours, make an arbitrary choice.

3. Introduce a new vertex v_x and add an edge between v_x and each common neighbour of e.

4. Delete e and the edges from v_l and v_r (the vertices at the ends of e), to the common neighbours of e.

5. Delete v_l and v_r and the remaining edges connected to them. These edges do not connect e to a common neighbour. The number of edges deleted in this step is n_d in step 1.
   Vertex v_x is now considered to represent a clique combining v_l and v_r.

6. If vertex v_x is isolated, GO TO step 1. Otherwise, compute n_d for each edge connected to v_x.

7. Choose the edge e, connected to v_x, with minimum n_d. If there is more than one such edge with minimum n_d, select the edge with most common neighbours. If more than one candidate edge has the largest number of common neighbours, make an arbitrary choice.

8. GO TO step 3.

Figure 2.3: The maximum included–edge clique partition algorithm.

1. If all vertices have zero degree, GO TO step 7.
2. Choose a non–zero degree vertex with the smallest degree. Call this $N1$.
3. Of all the vertices connected to $N1$, choose the one with the smallest degree. Call this $N2$. If more than one vertex is identified as a candidate for $N2$, choose one that has a common neighbour with the edge$(N1, N2)$. If no candidate has a common neighbour with $(N1, N2)$, make an arbitrary choice.
4. Cluster $N1$ and $N2$ into a new compound vertex $N3$.
5. Modify the graph by replacing vertices $N1$ and $N2$ with the new vertex $N3$. A vertex Nx is connected to $N3$ in the new graph *iff* Nx was connected to both $N1$ and $N2$ in the old graph. The degree of affected vertices is updated.
6. GO TO step 1.
7. Make any vertices not already clustered into singular clusters. STOP.

Figure 2.4: The Bhasker–Samad (METHOD–2) clique partition algorithm.

1. Pick the edge (p, q) which has the maximum number of common neighbours (a vertex is a common neighbour of an edge if it is connected to both vertices of the edge). If the graph has no edges then STOP. If multiple edges have the same maximum number of common neighbours, then choose the edge which would result in the deletion of the fewest number of edges (see Step 3).
2. Cluster p and q into a clique.
3. Modify the graph by replacing p and q by a new vertex r. A vertex v is connected to r in the new graph *iff* v was connected to both p and q in the old graph. (All edges connecting p with a vertex to which q was not connected, and all edges connecting q with a vertex to which p was not connected, are deleted.)
4. If vertex r is isolated, GO TO 1. Else pick an edge s which includes r as a vertex and which has the maximum number of common neighbours. If multiple edges meet this criterion, choose the edge which would result in the deletion of the fewest number of edges.
5. Rename r and s as p and q.
6. GO TO 3.

In steps 1 and 4, if multiple edges meet the conditions, an arbitrary choice is made.

Figure 2.5: The Tseng clique partition algorithm.

This algorithm has some advantages over the Tseng [42] and Bhasker–Samad [40] algorithms for this application (it must be noted that neither of these algorithms were developed to perform well for the maximum included edge problem specifically). In particular, Tseng's algorithm may require an arbitrary choice of edges to be made during execution, regardless of the effect on the number of edges included in cliques. The Bhasker–Samad algorithm considers least–connected vertices first and so can construct small cliques which prevent larger cliques connected to them from reaching full size.

Figure 2.6 illustrates the operation of the proposed algorithm and the algorithms of Tseng and of Bhasker *et al* on a small test graph. The Bhasker–Samad algorithm first constructs cliques incorporating the lowest–degree vertices and so increases the number of edges between cliques for this graph.

The performance of the three algorithms was compared by applying them to suites of 100 random simple connected graphs of 100 vertices with varying density. All three algorithms were implemented in C++ using the LEDA class library [43]. Table 2.1 shows their average performance. A "good" algorithm would minimise the number of cliques in the partition and simultaneously maximise the number of edges contained within cliques. The Bhasker–Samad algorithm generally produces clique partitions with fewer cliques than the Tseng algorithm does. On the other hand, the Tseng algorithm includes more edges within cliques. On this suite of test graphs, the MIECP algorithm usually performs marginally better than both of them in simultaneously maximising the number edges contained within cliques and minimising the number of cliques.

Table 2.2 shows the performance of the three algorithms on a second suite of random graphs. In this case, test graphs of 100 vertices were constructed by selecting a set of clique sizes from a normal distribution with specified mean and standard deviation. The cliques were then interconnected by randomly selected edges so that the overall graph was simple and connected. The table shows the effect of varying the density of the corresponding compound graph, that is, the graph whose vertices represent the set of cliques. Here, a density of zero corresponds to connecting the cliques in a random tree, while a density of one corresponds to adding an edge between all pairs of cliques. The Tseng algorithm and the MIECP algorithm have virtually identical performance, and both perform somewhat better

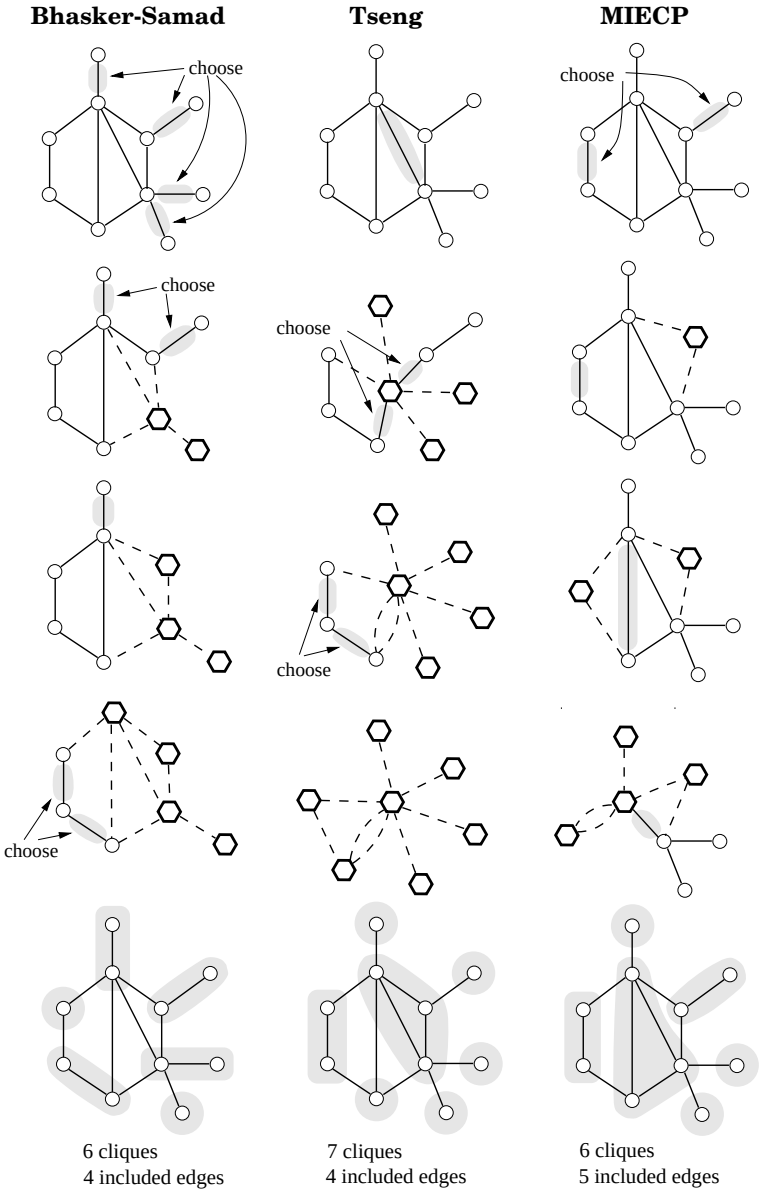


Figure 2.6: Comparison of steps in the operation of the three clique partition algorithms discussed.

Table 2.1: Performance of the clique partition algorithms on random connected graphs with 25, 50 and 100 vertices, averaged over 100 trials.

ρ	$\lvert E\rvert$	$\overline{\rho_{final}}$			Number of Cliques			Included Edges		
		B–S	TSENG	MIECP	B–S	TSENG	MIECP	B–S	TSENG	MIECP
25 vertices										
.1	30	.21	.20	.20	13.5	13.7	13.7	11.6	12.3	12.3
.2	60	.56	.50	.53	11.3	11.6	11.3	16.3	18.5	18.6
.3	90	.82	.72	.75	9.66	9.84	9.60	21.8	24.8	25.2
.4	120	.96	.85	.88	8.25	8.50	8.37	27.7	31.6	31.8
.5	150	.99	.91	.94	7.15	7.47	7.18	34.1	39.8	40.2
.6	180	1.00	.95	.96	6.10	6.37	6.23	42.4	49.4	49.7
.7	210	1.00	.98	.98	5.11	5.33	5.19	53.4	63.0	63.2
.8	240	1.00	.99	.99	4.10	4.33	4.25	70.7	82.8	83.7
.9	270	1.00	.99	.99	3.05	3.32	3.26	106.0	123.1	123.2
50 vertices										
.1	122	.31	.28	.30	23.6	23.9	23.5	29.5	33.0	33.2
.2	245	.70	.65	.70	19.2	18.9	18.3	43.3	49.5	49.7
.3	367	.93	.84	.91	16.1	16.0	15.1	56.6	65.1	66.5
.4	490	.99	.92	.97	13.4	13.7	12.8	72.3	82.8	84.2
.5	612	1.00	.96	.99	11.3	11.6	10.9	90.9	104.9	106.6
.6	735	1.00	.98	.99	9.37	9.89	9.35	115.2	132.2	132.9
.7	857	1.00	.99	1.00	7.75	8.14	7.66	148.0	171.1	172.7
.8	980	1.00	.99	1.00	5.98	6.40	6.05	202.2	233.3	236.3
.9	1102	1.00	1.00	1.00	4.14	4.63	4.35	309.5	361.1	363.3
100 vertices										
.1	495	.39	.40	.43	42.5	40.2	39.0	73.0	85.2	86.1
.2	990	.82	.78	.83	33.1	31.6	30.6	106.8	125.1	125.2
.3	1485	.98	.91	.96	27.2	26.5	25.1	140.5	164.0	165.2
.4	1980	1.00	.97	.99	22.7	22.2	21.0	178.9	207.9	211.0
.5	2475	1.00	.98	1.00	18.6	18.6	17.5	227.8	264.5	267.4
.6	2970	1.00	.99	1.00	15.2	15.4	14.6	291.4	337.6	340.1
.7	3464	1.00	1.00	1.00	12.1	12.4	11.6	383.4	444.0	447.9
.8	3960	1.00	1.00	1.00	9.08	9.59	9.01	534.3	616.8	620.8
.9	4454	1.00	1.00	1.00	6.01	6.50	6.11	869.7	1001.4	1008.0

Abbreviations: ρ: density of input graph, $\overline{\rho_{final}}$: density of final simple graph, B–S: Bhasker–Samad algorithm (METHOD–2), TSENG: Tseng algorithm, MIECP: maximum included–edge algorithm.

Table 2.2: Performance of the clique partition algorithms, averaged over 100 trials, on random connected graphs of 100 vertices, constructed as described in the text.

d	Number of Cliques			Number of Included Edges			Number of Residual Edges		
	B–S	TSENG	MIECP	B–S	TSENG	MIECP	B–S	TSENG	MIECP
Average clique size = 5, standard deviation = 2									
0	43.61	18.56	18.56	177.6	257.7	257.7	97.7	17.6	17.6
.2	18.20	18.17	18.17	257.8	259.1	259.1	46.3	44.9	44.9
.4	18.38	18.33	18.34	257.1	259.0	259.0	76.0	74.0	74.1
.6	18.89	18.62	18.62	250.0	253.9	253.9	109.9	106.0	106.1
.8	18.85	18.49	18.49	253.7	257.6	257.5	138.5	134.6	134.7
1	18.68	18.38	18.38	253.2	257.4	257.4	165.1	160.8	160.8
Average clique size = 10, standard deviation = 4									
0	15.13	9.80	9.80	505.1	536.8	536.8	40.53	8.80	8.80
.2	9.64	9.64	9.64	542.7	543.7	543.7	16.03	15.05	15.05
.4	9.62	9.62	9.62	539.3	539.7	539.7	22.13	21.67	21.67
.6	9.88	9.88	9.88	528.7	530.0	530.0	31.28	29.96	29.96
.8	9.59	9.59	9.59	547.5	548.6	548.6	36.35	35.22	35.22
1	9.52	9.52	9.52	551.2	552.3	552.3	42.43	41.28	41.28
Average clique size = 20, standard deviation = 8									
0	7.87	5.06	5.06	1069.6	1099.9	1099.9	34.40	4.06	4.06
.2	6.06	5.11	5.11	1061.0	1074.1	1074.1	18.34	5.22	5.22
.4	5.12	5.12	5.12	1077.3	1077.3	1077.3	6.63	6.63	6.63
.6	5.06	5.06	5.06	1102.7	1103.3	1103.3	8.40	7.87	7.87
.8	5.03	5.03	5.03	1089.5	1089.5	1089.5	9.22	9.22	9.22
1	4.94	4.94	4.94	1106.8	1107.3	1107.3	10.67	10.17	10.17

Abbreviations: d: density of compound graph ($d = 0.0$ corresponds to a tree and $d = 1.0$ to a complete graph), B–S: Bhasker–Samad algorithm (METHOD–2), TSENG: Tseng algorithm, MIECP: maximum included–edge algorithm.

than the Bhasker–Samad algorithm at low densities.

2.3.3 Time Complexity

A simple estimate of the time complexity of the MIECP algorithm can be made by assuming that the graph G with N vertices is represented by an adjacency matrix. In Step 1 of the algorithm, each edge must be examined and its common neighbours identified. (A common neighbour of edge e is a vertex which is connected by an edge to both the vertices v_l, v_r at the ends of e.) This can be in $O(N)$ time for each edge by AND–ing the two rows of the adjacency matrix representing vertices v_l and v_r. The number of edges, n_d, which would be deleted if this edge were chosen for inclusion in a clique, is given by

$$n_d = d_l + d_r - 2 - 2N_{cn}(e)$$

where

d_l and d_r are the degrees of v_l and v_r respectively,

and $N_{cn}(e)$ is the number of common neighbours of e.

Computation of the vertex degrees also takes $O(N)$ time for each edge. Since there are $O(N^2)$ edges, Step 1 takes $O(N^3)$ time. Selection of the edge with minimum n_d and maximum number of common neighbours in Step 2 then takes $O(N^2)$ time. Step 3 adds a maximum of N edges and so is $O(N)$ time, and Steps 4 and 5 delete a maximum of $2N$ edges in $O(N)$ time. Step 6 requires examining $O(N)$ edges, each requiring $O(N)$ time to compute n_d and is thus $O(N^2)$ time. Finally, Step 7 must examine $O(N)$ edges to find the next candidate for inclusion in a clique.

At each iteration of the algorithm, the number of vertices decreases by 1, and so there are $O(N)$ total iterations. In the least favourable case, where step 1 has to be executed at each iteration, the total time complexity is thus $O(N^4)$.

In this extreme case, the algorithm recalculates n_d for each edge at each iteration, taking $O(N^3)$ time. An improvement in speed could be made by maintaining a record of the n_d for each edge, conveniently by storing $n_d + 1$ in the adjacency matrix. When an selected edge is included in a clique, this matrix can be updated at a cost $O(N^2)$ time with the algorithm shown in Figure 2.7.

Step 1 of the update operation requires AND–ing two rows of the adjacency matrix and thus takes $O(N)$ time. Step 2 examines all edges incident to the end–vertices of e and is thus $O(N^2)$ time. Step 3 requires the examination of $O(N^2)$ entries in the adjacency matrix and thus takes $O(N^2)$ time. Step 4 takes $O(N)$ time and the computation of the n_d values for the new vertex in Step 5 takes $O(N^2)$ time. Overall, the update operation thus takes $O(N^2)$ time. The $O(N)$ total iterations needed in the modified MIECP algorithm with update results in a time complexity $O(N^3)$.

By comparison, the basic Bhasker–Samad algorithm (Method–2) requires $O(N^3)$ time, mainly because it operates by selecting vertices rather than edges. Bhasker

1. Decrement the n_d value of all the common neighbours of e.
2. For all other vertices v (not common neighbours) connected to the end–vertices of e, decrement the n_d value of all edges incident to v.
3. For all vertices v (not common neighbours) connected to the end–vertices of e check whether there is an edge incident to v which has an end–vertex of e as a common neighbour. If so, increment its n_d value by 2.
4. Zero the rows and columns corresponding to the end–vertices of edge e.
5. Add a row and column to the adjacency matrix to represent the new vertex. This can be done in the space freed in step 4.

Figure 2.7: Operations required to update the adjacency matrix.

et al also propose a data structure to improve the speed of their algorithm. The time complexity of the basic Tseng algorithm is $O(N^4)$.

2.4 Remarks

Graph simplification is one approach to coping with the difficulties of drawing dense graphs in a limited display region. Grouping vertices together, on the basis of either some common property in the application domain, or a graph–theoretic property, provides an abstraction mechanism for simplifying the graph. Grouping vertices into cliques is only one of a number of possibilities. As part of a practical general visualisation system, a useful graph simplification tool would offer a range of user– and automatically–selected techniques.

Chapter 3

The 3D Spring Algorithm

The spring algorithm [20] has a venerable place in the relatively short history of graph layout. It belongs to the class of physics–based methods, which draw a formal analogy between the layout problem and the behaviour of some idealised physical system which is easy to simulate computationally.

The spring algorithm is easy to understand and implement, and it generally produces pleasing layouts which display the symmetries of the graph [44]. Although originally proposed for two-dimensional layout of general undirected graphs it has subsequently been applied to the 3D layout problem [45, 46], and as a component of more specialised algorithms (see Chapter 4).

The conformance criteria for effective graph layouts discussed in Chapter 1, and user preferences, can often be implemented as mechanical constraints. Our limited experience with layouts in 3D also suggests that constraining the vertices to lie on simple 3D surfaces can improve the *readability* of the layout. The inclusion of such mechanical constraints on the system will be discussed below.

3.1 Background

The idea behind the spring algorithm is simple and intuitive. The input graph is used to define a mechanical system of ideal frictionless hinges, connected by springs. Each vertex in the graph is represented by a hinge, and each edge by a spring ("type I") of specified natural or unstressed length. Additional springs ("type II"), always repulsive, are placed between all vertex pairs not connected by

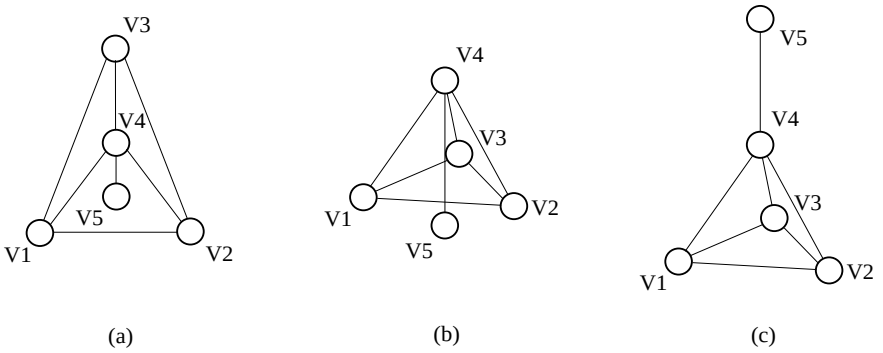


Figure 3.1: The graph shown in (a) has the two distinct equilibrium configurations (b) and (c) in three dimensions.

an edge. The spring algorithm finds an equilibrium configuration of this idealised mechanical system, and that configuration is taken to be the desired layout.

Eades [20] pointed out that at equilibrium, type I springs, representing edges, tend to equalise in length, and type II springs tend to separate vertices not connected by an edge, thus approximately satisfying two important drawing aesthetics

- all edges of the graph should be drawn with the same length, and
- vertices which do not have an edge between them should not be drawn close together.

A further valuable property of the algorithm is that symmetries in the graph tend to be displayed in the equilibrium configuration [44].

An equilibrium configuration of the mechanical system can be characterised by two equivalent conditions: at equilibrium, the forces acting on each vertex exactly cancel (in the following, "vertex" and "hinge" will be taken to be synonymous), and the potential energy stored in the springs is stationary (i.e. has zero derivative) with respect to the vertex positions (for a proof of this equivalence, see [47]). These conditions suggest the two main techniques for finding equilibria, given an arbitrary initial configuration. The first leads to "force–minimisation" methods, which try to find configurations in which the total force on each vertex is zero, and the second leads to "energy–minimisation" methods, which seek a configuration minimising the stored spring energy or some generalisation of it.

In general, the mechanical system has a number of different equilibrium configurations (Figure 3.1 shows an example). Finding the set of all equilibria (for

example to identify the "best" one for display) is currently largely a manual search procedure. Successful but expensive approaches have been to try different (random) initial configurations or to exchange the positions of vertex pairs followed by a further search for an equilibrium with a global minimum of stored energy [9].

Many refinements and extensions of the basic search algorithms have been suggested. Computational speed improvements have been made in the force-directed methods by using simpler force laws, integer arithmetic and by restricting the range of the type II springs [45]. Additional forces have been included to improve and constrain the drawing (edge-edge and edge-vertex forces [48], "magnetic" alignment of edges [48, 49], constraint of vertices to lie in planes [48] and within bounding boxes [45, 48]).

The energy minimisation method proposed by Kamada [9, 50] uses a classical Newton-Raphson numerical extremum search technique [51] to find an equilibrium configuration of a linear-spring system. Additional drawing constraints lead to a *constrained* minimisation problem. The simplest numerical approach to solving such problems is to minimise a cost function comprising the stored spring energy and additional terms which represent the penalty associated with not satisfying a particular constraint. Classical methods such as gradient descent have been applied to this problem [52], as well as simulated annealing [53, 54, 55, 56, 57] and genetic algorithms [58, 59] to permit the inclusion of more complex drawing constraints such as the minimisation of edge crossings. A parallel algorithm for increasing the execution speed of the simulated annealing optimisation was presented in [46].

Force-directed methods proceed by initially placing the vertices in an arbitrary configuration and then numerically simulating the relaxation of the mechanical system towards an equilibrium. The residual force acting on each vertex is used to compute an incremental modification of the vertex positions. This procedure in effect carries out a numerical solution of the system of ordinary differential equations (ODE's) describing the evolution of the mechanical system.

This chapter is an elaboration of the ODE formulation of the spring algorithm. The formulation shows that equilibrium configurations can be found with classical Numerical Analysis methods which can considerably improve execution times over existing methods. Further, it shows that many constraints such as those proposed

in Chapter 1 can be readily incorporated with the spring algorithm.

3.2 Formulating the Mechanical Model

The spring algorithm is applicable to undirected, connected simple graphs. The graphs are not required to have any special properties beyond connectedness. General directed graphs can of course also be drawn by ignoring the edge directions.

Given a graph G with vertex set V and edge set E, a mechanical model is defined by introducing an ideal hinge to represent each vertex $v \in V$. The position of a hinge in the model is taken to be the position of the corresponding vertex in the layout.

The edges $e \in E$ are represented by springs. These springs ("type I") are usually identical for all edges and have some natural, unstrained length. The intention is to apply an attractive force when the edge length is greater than the natural spring length, and a repulsive force when the edge length is smaller. The spring "force law", that is, the relationship between the magnitude of the force and the spring length, is monotonic and zero at the natural length.

The precise form of the force law can be chosen to improve the final layout. The classical ideal linear spring law (Hooke's law) may be too strong in the attraction region, and Eades[20] has suggested a logarithmic law. For a reduction in computational effort, a rational polynomial law has also been suggested [45].

$$f(s) = \begin{cases} k_c(L-s) & \text{linear (Hooke)} \\ k_c \ln(\frac{L}{s}) & \text{logarithmic} \\ k_c(L^2/s - s^2/L) & \text{rational polynomial} \end{cases}$$

where

s is the spring length,

L is the unstrained spring length,

$f(s)$ is the magnitude of the force developed when the spring has length s, and

k_c is a constant.

A repulsive force is also applied between all vertices not directly connected by an edge. This is implemented as a spring ("type II"), usually with an inverse-square law:

$$f(s) = k_u \left(\frac{L}{s}\right)^2 .$$

Kamada [9] used type I springs to provide this repulsive force by setting their natural length to the shortest graph-theoretic path length between the respective vertices.

As mentioned above, the ODE formulation of the spring algorithm attempts to simulate the relaxation of this mechanical model of the graph from some arbitrary initial configuration to a stable equilibrium. At an equilibrium, the total energy stored in the springs of the model is at a stationary point, that is, the derivative of the total stored energy with respect to the vertex positions is zero. The energy difference between this point and the total energy in the initial configuration must be dissipated in some way for the model to converge to, and remain at, an equilibrium configuration.

The simplest form of energy dissipation is provided by a restraining force proportional to velocity and acting in the opposite direction. This is the classical model of ideal viscous damping. Newton's second law applied to this system gives the basic system of second order ODE's completely describing the motion of the vertices:

$$m\frac{d^2\mathbf{r}_k}{dt^2} \equiv m\ddot{\mathbf{r}}_k = \mathbf{F}_k \qquad k = 1, \ldots, N, \tag{3.1}$$

where

N is the number of vertices in the graph being modeled,

m is the mass of each hinge,

$\mathbf{r}_k$ is the position vector of vertex v_k,

t is time, and

$\mathbf{F}_k$ is the total force acting on vertex v_k.

Assuming ideal viscous damping at each vertex, this becomes

$$m\ddot{\mathbf{r}}_k = \mathbf{F}_k^s(\mathbf{r}_1, \ldots, \mathbf{r}_N) - D\dot{\mathbf{r}}_k \qquad k = 1, \ldots, N$$

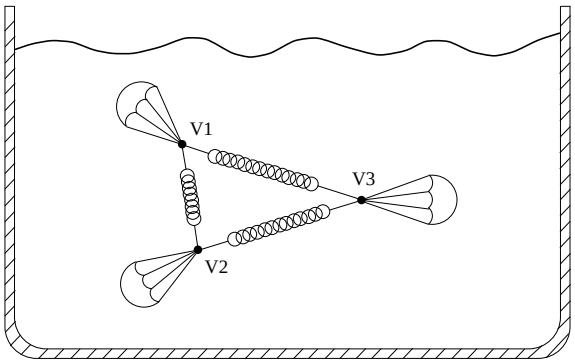


Figure 3.2: Graph layout modeled as a damped spring system.

where

$\mathbf{F}_k^s(\mathbf{r}_1, \ldots, \mathbf{r}_N)$ is the total force acting on vertex v_k due to the springs and constraints,

$\dot{\mathbf{r}}_k \equiv d\mathbf{r}_k/dt$ is the velocity of the hinge corresponding to vertex v_k, and

D is a constant defining the strength of viscous damping.

At an equilibrium, all forces are balanced and all the vertices are at rest, or $\ddot{\mathbf{r}}_k = \dot{\mathbf{r}}_k = \mathbf{0}, \quad k = 1, \ldots, N$. Thus the equilibrium configurations are the roots of the nonlinear system of equations

$$\mathbf{F}_k^s(\mathbf{r}_1, \ldots, \mathbf{r}_N) = \mathbf{0} \qquad k = 1, \ldots, N.$$

Because of the complexity of this system (e.g. $3N$ unknowns in 3D) it is not feasible to find the roots directly from these equations except in the simplest cases. However, a numerical solution of the system of ODE's displays the evolution of the mechanical system from some initial configuration to one of these equilibrium configurations.

In the graph layout application, it is the final equilibrium configuration that is of interest rather than details of the motion during relaxation, so we are free to choose the parameters m and D for convenience. In particular, the choice $m = 0$ gives the simpler system of *first* order ODE's

$$\dot{\mathbf{r}}_k = \frac{1}{D}\mathbf{F}_k^s(\mathbf{r}_1, \ldots, \mathbf{r}_N) \qquad k = 1, \ldots, N. \tag{3.2}$$

This describes the mechanical system suggested by Figure 3.2. Massless hinges are acted on by the combination of spring forces and a damping force represented in the figure by an oil bath. The simplest method for numerically calculating the motion of this system discretises the derivative:

$$\begin{aligned} \dot{\mathbf{r}}_k(t) &= \lim_{\delta t \to 0} \frac{\mathbf{r}_k(t+\delta t) - \mathbf{r}_k(t)}{\delta t} \quad k = 1, \ldots, N, \\ \text{so} \quad \dot{\mathbf{r}}_k(t) &\approx \frac{\mathbf{r}_k(t+\Delta t) - \mathbf{r}_k(t)}{\Delta t} \quad \text{for small} \quad \Delta t, \quad k = 1, \ldots, N. \end{aligned}$$

This is the basis of the well-known Euler method and gives the iteration

$$\begin{aligned} \mathbf{r}_k(t+\Delta t) &= \mathbf{r}_k(t) + \left(\frac{\Delta t}{D}\right) \mathbf{F}_k^s(\mathbf{r}_1(t), \ldots, \mathbf{r}_N(t)), \\ \text{or} \quad \mathbf{r}_{k,n+1} &= \mathbf{r}_{k,n} + \left(\frac{\Delta t}{D}\right) \mathbf{F}_k^s(\mathbf{r}_{1,n}, \ldots, \mathbf{r}_{N,n}). \end{aligned}$$

This can be compared with the commonly used "follow your nose" algorithm [20, 45, 48]

$$\mathbf{r}_{k,n+1} = \mathbf{r}_{k,n} + \epsilon \mathbf{F}_k^s(\mathbf{r}_{1,n}, \ldots, \mathbf{r}_{N,n}),$$

where ϵ is some small number, possibly varying during the motion. It states that at each iteration, every vertex should be moved a "small distance" $\epsilon \mathbf{F}_k^s$ in the direction of, and proportional to, the total force acting on it. In other words, this algorithm is equivalent to applying Euler's method with step size ϵD to a modified spring system of zero–mass vertices with viscous damping.

The consequences of using this simple numerical method for solving the system of ODE's will be discussed in a later section.

3.3 Qualitative Properties of the ODE System

The motion of the mechanical system is completely described by Equation 3.2. This is a vector system of first–order ordinary differential equations. It can be written in terms of the $3N$ (in 3D) vector components (in *normal form*) as

$$\dot{y}_i = f_i(y_1, y_2, \ldots, y_{3N}) \quad i = 1, \ldots, 3N \tag{3.3}$$

where N is the number of vertices in the graph, and

$$\left.\begin{aligned} y_{3i-2} &= \mathbf{r}_{ix} \\ y_{3i-1} &= \mathbf{r}_{iy} \\ y_{3i} &= \mathbf{r}_{iz} \\ f_{3i-2} &= \mathbf{F}_{ix} \\ f_{3i-1} &= \mathbf{F}_{iy} \\ f_{3i} &= \mathbf{F}_{iz} \end{aligned}\right\} \quad \text{for} \quad i = 1, \ldots, N$$

This is an *autonomous* [47] system, i.e. the time variable does not appear explicitly on the right hand side, and the functions f_i are non–linear. There are no known methods for finding analytic solutions of systems of this kind except in a few special cases. It is nevertheless possible to make some qualitative observations about the nature of the solutions.

Configurations for which $\dot{y}_i = f_i = 0$ are known as the *equilibrium, stationary* or *singular* points of the system. There must be at least one stationary point. This can be seen in a result from classical mechanics for a dissipative massless system[47]:

$$\frac{dU}{dt} = -2\mathcal{F},$$

where U is the potential energy of the system, $\mathcal{F}$ is Rayleigh's dissipation function [60]:

$$\mathcal{F} = \frac{1}{2}\sum_{i=1}^{3N} D\dot{y_i}^2.$$

Since $\mathcal{F}$ is non–negative, U must be monotonically decreasing and so reaches a (local) minimum as $t \to \infty$.

A particular configuration of the mechanical system can be viewed as a point $(y_1, y_2, \ldots, y_{3N})$ in a $3N$ dimensional Euclidean space (the *configuration* space). The evolution of the mechanical system traces a curve, or *orbit*, parametrised by the time t, from some initial point $\mathbf{y_i}$ to a stationary point $\mathbf{y_o}$.

In some region surrounding a stationary point $\mathbf{y_o}$, the nonlinear system can be approximated by the linear system

$$\dot{\mathbf{y}} = A\mathbf{y},$$

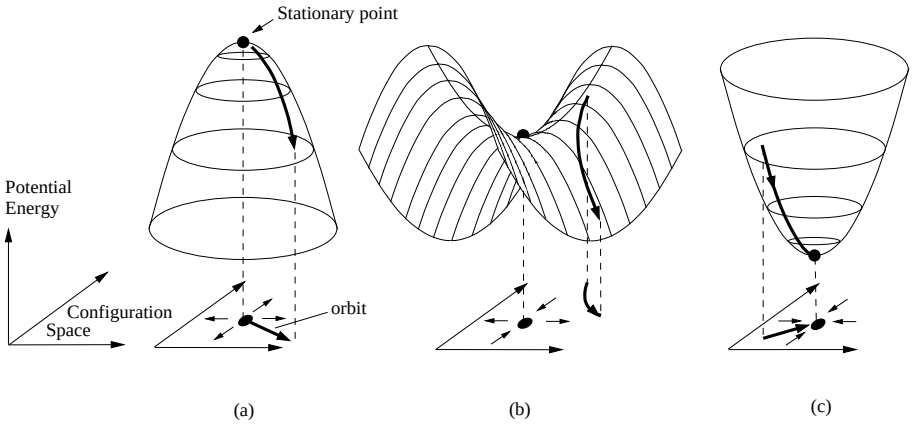


Figure 3.3: The local geometry of stationary points in the stored energy of spring systems: (a) unstable, (b) saddle, and (c) stable equilibria.

where the *Jacobian* matrix A is defined as

$$A = \begin{pmatrix} \frac{\partial f_1(\mathbf{y_o})}{\partial y_1} & \frac{\partial f_1(\mathbf{y_o})}{\partial y_2} & \cdots & \frac{\partial f_1(\mathbf{y_o})}{\partial y_{3N}} \\ \frac{\partial f_2(\mathbf{y_o})}{\partial y_1} & \frac{\partial f_2(\mathbf{y_o})}{\partial y_2} & \cdots & \frac{\partial f_2(\mathbf{y_o})}{\partial y_{3N}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_{3N}(\mathbf{y_o})}{\partial y_1} & \frac{\partial f_{3N}(\mathbf{y_o})}{\partial y_2} & \cdots & \frac{\partial f_{3N}(\mathbf{y_o})}{\partial y_{3N}} \end{pmatrix}.$$

The theory of linear ODE's [61] shows that there are essentially three kinds of stationary point, distinguished by the eigenvalues of A (see Figure 3.3). In the neighbourhood of *stable* equilibria, all orbits converge to the stationary point. Near *unstable* equilibria, either all orbits diverge from the stationary point, or for saddle points, almost all orbits avoid the stationary point. The stable equilibria are of interest in the graph layout problem. Due to numerical artefacts such as rounding errors, a numerical solution of Equation 3.3 is unlikely to converge to unstable equilibria.

The number of equilibria, and the partition of the configuration space into subspaces such that all starting configurations within a subspace have orbits leading to identical equilibrium configurations are open questions.

3.4 Constraints

The conformance and adaptability requirements for a graph drawing system (see Chapter 1) can often be expressed as additional constraints on the mechanical

model. The most commonly used constraints are

- fixed vertices (one or more vertices fixed in position),
- rigid edges (one or more edges are fixed in length),
- edge alignment (one or more edges have a specified direction, or tendency to align in some direction),
- enclosure or exclusion (one or more vertices are confined to or excluded from the interior of a prescribed region), and
- constraint to a curve or surface (one or more vertices or edges may be constrained to lie on a prescribed curve or surface) [8, 58, 9, 62, 63, 64, 65].

In this section, the incorporation of constraints of these kinds into the ODE formulation of the spring algorithm will be discussed.

Mechanical constraints are classified as *holonomic* or *non–holonomic* [47]. Holonomic constraints are constraints which can be written in the form

$$g(y_1, y_2, \ldots, y_{3N}) = 0. \tag{3.4}$$

For example, constraining a vertex with position $\mathbf{r}_i$ to lie on a sphere of radius R and centred at $\mathbf{r}_c$ can be expressed as

$$| \mathbf{r}_i - \mathbf{r}_c | \quad = R.$$

Fixed vertices, rigid or aligned edges and constraint of vertices to a curve or surface are all examples of holonomic constraints and can be treated by a general unified approach to be described below.

Non–holonomic constraints cannot be expressed in the form of Equation 3.4. For example, enclosure of a vertex within a sphere of radius R and centred at $\mathbf{r}_c$ can be expressed as

$$| \mathbf{r}_i - \mathbf{r}_c | \quad < R.$$

Non–holonomic constraints are generally difficult to implement. The simplest approach in the above example is to track the path of the vertex, preventing excursions across the boundary of the enclosing region [66]. However it is usually

sufficient in graph layout applications to approximate non–holonomic constraints by a combination of suitable forces. A repulsive force inversely proportional to the distance from an enclosing surface has been used successfully [45, 48] to constrain vertices to the interior of the surface.

The complete description of the motion of a mechanical system with holonomic constraints is a central achievement of classical mechanics [47, 67]. The use of Newton's formulation of the equations of motion (Equation 3.1) for a system with constraints would require knowledge of all the forces caused by the constraints. For example, the equations describing the motion of a vertex constrained to move on the surface of a sphere require an additional force term describing the reaction force of the sphere on the vertex.

The use of the *Lagrangian* formulation of the equations of motion avoids the necessity of determining these constraint forces. Lagrange's equations describing the spring system are [47]:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{y_i}}\right) = \frac{\partial L}{\partial y_i}, \quad i = 1, \ldots, N$$

where

$$\begin{aligned} L &= \text{Kinetic Energy} - \text{Potential Energy} && (3.5)\\ &= \frac{1}{2}\sum_{i=1,3N} m_i \dot{y_i}^2 - U(y_1, y_2, \ldots, y_{3N}). && (3.6) \end{aligned}$$

Holonomic constraints on the mechanical system are incorporated by using the constraint equations (3.4) to reduce the number of degrees of freedom in the system. Assume there are M holonomic constraints expressed as

$$g_i(y_1, y_2, \ldots, y_{3N}) = 0, \quad i = 1, \ldots, M.$$

These can be used to eliminate M of the $3N$ coordinates $\{y_i\}$. In terms of $3N - M$ new *generalised coordinates* $\{q_i\}$

$$y_i = y_i(q_1, q_2, \ldots, q_{3N-M}), \quad i = 1, \ldots, 3N.$$

The Lagrange equations with the inclusion of ideal viscous drag become

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q_i}}\right) - \frac{\partial L}{\partial q_i} + \frac{\partial \mathcal{F}}{\partial \dot{q_i}} = 0 \quad i = 1, \ldots, 3N - M, \qquad (3.7)$$

where $\mathcal{F}(\dot{q_1}, \dot{q_2}, \ldots, \dot{q_{3N-M}})$ is Rayleigh's dissipation function.

Note that the dimensionality has been reduced by M, thus simplifying the problem, and that the constraints do not explicitly appear. A point with coordinates $(q_1, q_2, \ldots, q_{3N-M})$ represents a configuration of the system satisfying the constraints. The set of these points is a $3N - M$ dimensional sub–manifold (the *configuration space*) of the 3N dimensional Euclidean space of the positions $(y_1, y_2, \ldots, y_{3N})$, and the $(\hat{\mathbf{q}}_\mathbf{1}, \hat{\mathbf{q}}_\mathbf{2}, \ldots, \hat{\mathbf{q}}_{\mathbf{3N-M}})$ form a coordinate system on the constraint surface. A solution of Equations 3.7 is a path in the configuration space from some initial point to an equilibrium configuration.

For the massless system described by Equation 3.2, Equation 3.5 becomes

$$L = -U(q_1, q_2, \ldots, q_{3N-M}),$$

and so the Lagrange equations become

$$\frac{\partial U}{\partial q_i} + \frac{\partial \mathcal{F}}{\partial \dot{q_i}} = 0 \quad i = 1, \ldots, 3N - M,$$

or

$$\dot{q_i} = -\frac{1}{D}\frac{\partial U(q_1, q_2, \ldots, q_{3N-M})}{\partial q_i}.$$

The partial derivative on the right hand side is just the component of the force $\mathbf{F_i^s}$ acting in the direction $\hat{\mathbf{q}}_\mathbf{i}$ and so this has the same form as Newton's equations of motion for the massless system with viscous damping (Equation 3.2) but expressed in terms of motion in the configuration space.

Thus the motion of such a system with the constraint of one or more vertices to a smooth surface can be determined by (1) initially placing the constrained vertices somewhere on the surface, and (2) solving the system of equations 3.2 with the forces $\mathbf{F_i^s}$ projected onto a plane tangent to the surface at the positions of the constrained vertices (see Figure 3.4).

That is, the differential equations for the constrained vertices become

$$\dot{\mathbf{r}}_k = \frac{1}{D}\left[\mathbf{F}_k^s - (\mathbf{F}_k^s \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}\right], \quad k \in K,$$

where

$\mathbf{F}_k^s$ is the total spring force acting on vertex v_k, and

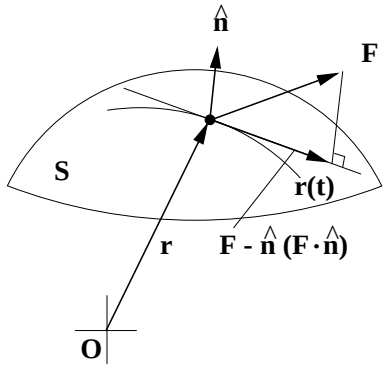


Figure 3.4: Projection of force to constrain a vertex to an arbitrary smooth surface.

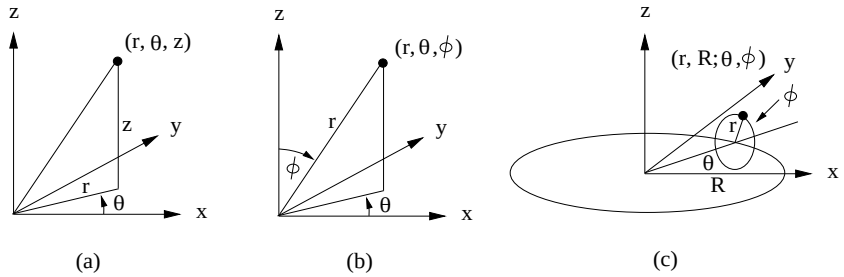


Figure 3.5: Coordinate systems for simple constraint surfaces: (a) cylindrical, (b) spherical and (c) toroidal.

$\hat{\mathbf{n}}$ is the unit normal vector to the surface at $\mathbf{r}_k$.

Different vertices can easily be assigned to distinct surfaces in this way. Four surface types have been implemented to demonstrate the approach: the plane, sphere, cone and torus. Constraint to curved lines is applied in Chapter 4 to layered drawings in 3D. The only significant difference is in the evaluation of the normal vector $\hat{\mathbf{n}}$ defining the plane tangent to the surface in the appropriate coordinate system (see Figure 3.5), as described below.

Line or curve Constraint to an arbitrary smooth curve is a degenerate case in which there is a normal *plane* at each point on the curve. This normal plane is perpendicular to the tangent vector $\hat{\mathbf{t}}$ to the curve at the point. Then the equations of motion become

$$\dot{\mathbf{r}}_k = \frac{1}{D}\left(\mathbf{F}_k^s \cdot \hat{\mathbf{t}}\right)\hat{\mathbf{t}}, \quad k \in K,$$

where

$\mathbf{F}_k^s$ is the total spring force acting on vertex v_k, and

$\hat{\mathbf{t}}$ is the unit tangent vector to the curve at $\mathbf{r}_k$.

If the curve is a straight line, $\hat{\mathbf{t}}$ is of course just a unit vector in the direction of the line.

Plane In this case the normal $\hat{\mathbf{n}}$ is a constant, perpendicular to the plane.

Sphere If the coordinates of a vertex on a sphere of radius R, centred on the origin are (R, θ, ϕ), the unit normal $\hat{\mathbf{n}}$ at the vertex position is given by

$$\hat{\mathbf{n}} = \begin{pmatrix} \sin\phi\cos\theta \\ \sin\phi\sin\theta \\ \cos\phi \end{pmatrix}.$$

Cone or cylinder For the cone with base centred in the x-y plane, the angle ϕ corresponds to the cone half-angle (0° for a cylinder) and $\hat{\mathbf{n}}$ is given by

$$\hat{\mathbf{n}} = \begin{pmatrix} \cos\phi\cos\theta \\ \cos\phi\sin\theta \\ \sin\phi \end{pmatrix}.$$

The discontinuity in the normal $\hat{\mathbf{n}}$ at the cone tip can be handled by suppressing any component of force in the positive z-direction.

Torus For the torus with line of symmetry along the z-axis and with vertex position defined by θ and ϕ,

$$\hat{\mathbf{n}} = \begin{pmatrix} \cos\phi\cos\theta \\ \cos\phi\sin\theta \\ \sin\phi \end{pmatrix}.$$

The other holonomic constraints mentioned earlier can be incorporated in this framework by expressing them as additional forces acting on the vertices. A rigid (fixed-length) edge between vertices at $\mathbf{r_1}$ and $\mathbf{r_2}$ can be expressed by the constraint

$$|\mathbf{r_2} - \mathbf{r_1}| = l_{12},$$

where l_{12} is the desired length. This constraint can be approximated by increasing the edge-spring constant k_c arbitrarily for this edge.

The approximate alignment of edges can be carried out by applying a torque to the edge [49], suggested by analogy with the tendency of a permanent magnet to align with an external magnetic field. If an undirected edge between vertices at $\mathbf{r_1}$ and $\mathbf{r_2}$ is to be aligned with a preferred direction $\hat{\mathbf{H}}$, an appropriate torque is given by

$$\mathbf{T} = \alpha \mathbf{sign}\left[(\mathbf{r_2} - \mathbf{r_1}) \cdot \hat{\mathbf{H}}\right] (\mathbf{r_2} - \mathbf{r_1}) \times \hat{\mathbf{H}},$$

where α is a constant. This torque is applied as equal and opposite forces on the vertices:

$$\begin{aligned} \mathbf{F}_1 &= -\frac{1}{2}\mathbf{T} \times (\mathbf{r_2} - \mathbf{r_1}), \\ \mathbf{F}_2 &= \frac{1}{2}\mathbf{T} \times (\mathbf{r_2} - \mathbf{r_1}). \end{aligned}$$

These forces tend to rotate the edge towards the preferred orientation through the smallest angle without altering the edge length.

Finally, the constraint of an *edge* to lie in a surface requires the determination of the shortest path (or *geodesic*) on the surface between the vertices it connects. Given the positions of the vertices and the geodesic curve between them, the vector spring force can be computed based on the edge length along the surface and the edge direction at the vertices (the tangent to the constrained edge at some point always lies in a tangent plane to the constraint surface at the point). A simple approximation may be made by *subdividing* the edge to be constrained, that is by replacing the edge by a chain of dummy vertices, and requiring the dummy vertices to lie on the constraint surface, while allowing the edge segments to be straight lines. This yields a piecewise-linear approximation to the geodesic curve on the surface. Figure 3.15 shows this approach for the complete graph K_4 drawn on a sphere using this approach. The graph vertices are V1, V2, V3 and V4, and the edges have been subdivided by adding five dummy vertices to each.

3.5 Numerical Solution and Stiffness

We now return to the question of numerically solving the system of ODE's (Equation 3.3 or Equation 3.7) describing the evolution of the mechanical system from

some initial configuration to a stationary point. Ideally, a numerical integration would relax the configuration to within some small neighbourhood of a stationary point in as few iterations as possible. However, ODE systems of the type being considered can exhibit a property known as *stiffness* [68, 69, 70]. Numerical solution of a stiff system of ODE's using a standard solution technique (of which Euler's method is the simplest example) is characterised by at best an excessively large number of iterations and at worst numerical instability leading to wild oscillations in the solution.

The development of efficient numerical solution methods for stiff systems ("stiff methods") is an active area of research (see for example [71, 72]). Stability analyses of solution methods [73] have given considerable insight into the problem, but there is still some controversy over its correct mathematical definition. One (somewhat tongue–in–cheek) proposal is surprisingly pragmatic: "a system of ODE's should be considered stiff when its numerical solution with stiff methods is faster, usually tremendously faster, than with non-stiff methods" [73].

There are however well-known indicators of the likelihood of stiffness in a system of ODE's and unfortunately some are evident in the equations describing the spring model. In fact, stiffness was first recognised in numerical studies of variable-rate spring systems [69, 74].

Stiffness is commonly noticed in the numerical solution of very stable, strongly damped systems and is particularly likely to appear as the solution approaches an equilibrium rather than during the initial transient phase. Another indication is the presence of widely separated time scales in the components of a solution. Equivalently, the eigenvalues of the Jacobian matrix (Equation 3.3) have negative real parts and the ratio of the largest to smallest real parts is large (this ratio is called the *stiffness ratio*).

These conditions can occur in the spring system. The forces used to model edges, repulsion between unadjacent vertices, and constraints may have widely different relative scales. In particular, it is often desirable to make the repulsive force between unadjacent vertices much weaker then the edge force to minimise distortions in the edge lengths displayed by the layout. The effect of varying the strength of the repulsive force is shown in Figures 3.6 and 3.7. Removal of the perceived distortion requires a reduction of the spring constant k_u from 0.02 to

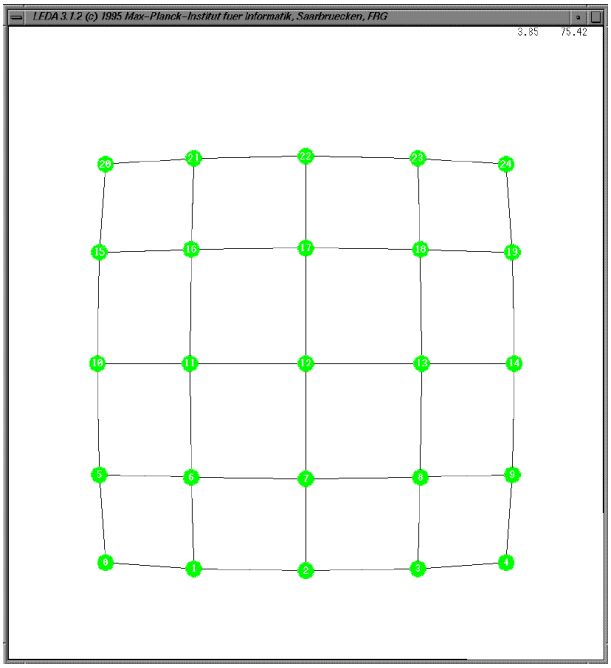


Figure 3.6: Drawing of a grid using a repulsive spring constant $k_u = 0.02$, and edge spring constant $k_c = 2$. The barrel distortion is caused by the repulsive forces (type II).

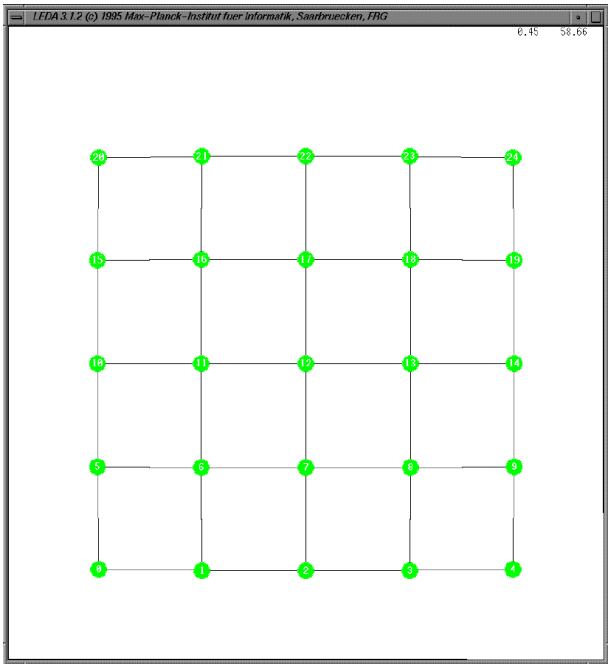


Figure 3.7: Drawing of a grid using a repulsive spring constant $k_u = 0.0002$, and edge spring constant $k_c = 2$.

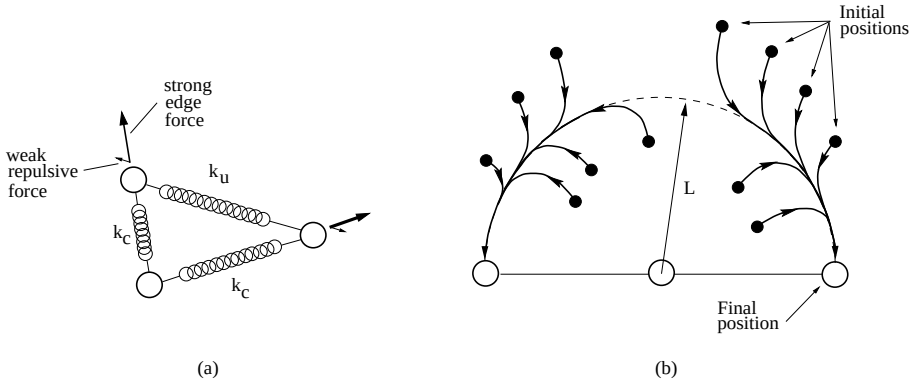


Figure 3.8: (a) Forces in a 3-vertex chain, (b) a weak repulsive force results in motions rapidly converging to the natural edge-spring length L, and then more slowly to the in-line equilibrium configuration.

0.0002 (with $k_c = 2$).

Under some circumstances, such a large difference in force constants causes components of the relaxation motion to have widely different time scales, leading to stiff behaviour in the numerical solution. Intuitively, this occurs when the forces can cause independent mechanical convergence. A simple example is the case of a graph of three vertices and two edges, i.e. with a central vertex having two adjacent vertices (see Figure 3.8).

The edge–modeling force (type I spring) causes a rapid convergence to a configuration where the outer vertices are equidistant from the central vertex, and nearly at the radius where the type I force is zero (L in the figure). The weaker repulsive force between the outer vertices causes the system to slowly straighten to the final stationary configuration where all three vertices are aligned. As shown schematically in Figure 3.8 (b), the outer vertices quickly approach the circle of radius L, regardless of the initial configuration, and then more slowly follow the circular arc to the in–line equilibrium configuration.

The effect can be analysed by examining the analytical solution of this simple example. The system can be simplified to the arrangement shown in Figure 3.9. For simplicity, the central vertex is considered to be fixed at the origin, the edge–modeling spring law is assumed to be linear, and we assume symmetry so that we need only consider the coordinates of one of the outer vertices (v_1). Using a linear spring with natural length L and spring constant k_c to represent the edges, an inverse-square law with constant k_u to represent the repulsive force between the

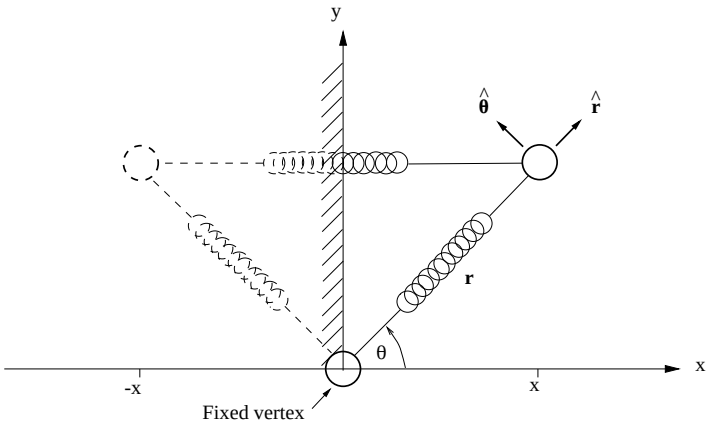


Figure 3.9: Geometry of the analytical solution of a 3-vertex chain

outer vertices, and a coefficient of viscous drag D, the ODE system describing the motion is simply

$$\dot{\mathbf{r}} = -\frac{k_c}{D}(r - L)\hat{\mathbf{r}} + \frac{k_u}{4Dx^2}\hat{\mathbf{x}}$$

where

r = $|\,\mathbf{r}\,|$,

$\hat{\mathbf{r}}$ = $\frac{\mathbf{r}}{r}$ is a unit vector in the $\mathbf{r}$ direction, and

$\hat{\mathbf{x}}$ is a unit vector in the x–direction.

It will be convenient to use polar coordinates (r, θ):

$$\begin{aligned} \mathbf{r} &= (r\cos\theta, r\sin\theta), \\ \text{so that } \hat{\mathbf{x}} &= \hat{\mathbf{r}}\cos\theta - \hat{\theta}\sin\theta \end{aligned}$$

The system Equation 3.5 then becomes

$$\begin{aligned} \dot{r} &= -\frac{k_c}{D}(r - L) + \frac{k_u}{4Dr^2\cos\theta} \\ \dot{\theta} &= -\frac{k_u \tan\theta}{4Dr^2\cos\theta}. \end{aligned}$$

The final equilibrium position is, to first order in k_u/k_c, $\theta = 0$, $r = L$. Changing variables to (ρ, θ), where $r = L + \rho$, and linearising the system near the equilibrium position ($\rho \ll L$, $\theta \ll 1$) the system becomes

$$\begin{aligned}\dot{\rho} &= -\frac{1}{D}\left(k_c + \frac{k_u}{2L^3}\right)\rho + \frac{k_u}{4DL^2} \\ \dot{\theta} &= -\frac{k_u\theta}{4DL^2}.\end{aligned}$$

To exhibit the eigenvalues of the Jacobian we make the further change of variable

$$\rho' = \rho + \frac{k_u}{4DL^2}\cdot\frac{D}{k_c + k_u/(2L^3)},$$

so that finally

$$\begin{aligned}\dot{\rho}' &= -\frac{1}{D}\left(k_c + \frac{k_u}{2L^3}\right)\rho' \\ \dot{\theta} &= -\frac{k_u\theta}{4DL^2},\end{aligned}$$

or

$$\begin{pmatrix} \dot{\rho}' \\ \dot{\theta} \end{pmatrix} = A \begin{pmatrix} \rho' \\ \theta \end{pmatrix} = \begin{pmatrix} -\frac{1}{D}\left(k_c + \frac{k_u}{2L^3}\right) & 0 \\ 0 & -\frac{k_u}{4DL^2} \end{pmatrix} \begin{pmatrix} \rho' \\ \theta \end{pmatrix}.$$

The eigenvalues of A are clearly

$$\begin{aligned}\lambda_1 &= -\frac{1}{D}\left(k_c + \frac{k_u}{2L^3}\right) \\ \lambda_2 &= -\frac{k_u}{4DL^2}.\end{aligned}$$

Thus the motion of the system near equilibrium is described by

$$\begin{aligned}\rho'(t) &= \alpha e^{-\lambda_1 t} \\ \theta(t) &= \beta e^{-\lambda_2 t},\end{aligned}$$

where α and β are constants depending on the initial vertex position. There are clearly two time scales ($1/\lambda_1$ and $1/\lambda_2$) and the stiffness ratio is

$$\frac{\lambda_1}{\lambda_2} = 4L^2\frac{k_c}{k_u} + \frac{2}{L}.$$

The stiffness ratio increases without bound as $k_u \to 0$. This is a predictor of stiff behaviour near the equilibrium position. A non-stiff ODE solution method

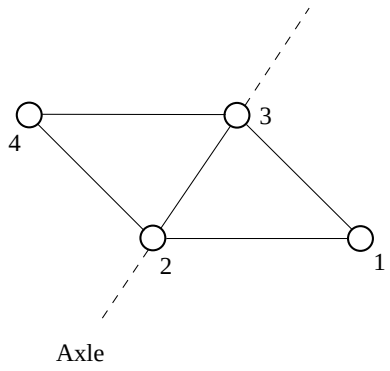


Figure 3.10: Stiffness effects typically appear in the numerical solution of a spring system when parts of the system can rotate without altering edge lengths. Here, vertices 1 and 4 can rotate (in 3D) about the "axle" formed by the edge between vertices 2 and 3.

must typically keep the step size smaller than the smallest time constant ($1/\lambda_1$ in this case) even though the corresponding component of the solution has decayed to negligible size.

This behaviour is also observed in larger systems when the mechanical model might be called *articulated*, that is, when it comprises two or more components which at equilibrium can be rotated relative to each other even if all edge springs were replaced by rigid rods. This happens in particular when the graph is not two–connected. In that case, the components can rotate around the cut-vertices under the influence of the repulsive and constraint forces alone. The idea of articulation as meant here is however slightly more general than lack of two–connectedness, as "axles" can also occur where two or more components of the model can hinge on a separator set of vertices (see Figure 3.10).

There are several options to manage this problem:

1. accept approximate equilibrium, i.e. halt the solution before equilibrium has been accurately reached (this seems to be the usual practice in force-directed implementations),

2. keep the repulsive force constant k_u large and accept distortions in the layout edge lengths,

3. alternate between two mechanical models in which (1) the repulsive force is inactive and (2) the edges are constrained to be rigid, thus avoiding the

simultaneous occurrence of widely separated time scales, or

4. use an ODE solution technique able to cope with stiff systems efficiently.

The development of a modern ODE solver for general stiff systems requires an effort measured in person-years, but fortunately such software is freely available [71, 72]. The approach outlined in this section has been implemented using the stiff-solver LSODA [75]. For greatest execution speed such algorithms generally require the computation of the Jacobian (Equation 3.3), although some packages optionally calculate a numerical approximation during execution. For the force laws appearing in the spring models discussed here it is straightforward to use algebraic manipulation packages such as REDUCE [76] to compute the components of the Jacobian matrix and even prepare compilable code directly.

As an example of the speed improvement in using a solution method specifically designed for stiff systems of ODE's, Table 3.1 shows the time taken to converge close to equilibrium using three methods at the same relative position error (0.00001) when applied to some graphs exhibiting stiffness. Convergence was assumed when the absolute magnitude of the residual forces acting on each vertex was < 0.001) A non-stiff method (*rkqs*, a 5th order Runge-Kutta method with adaptive stepsize control [77]) is compared with two stiff-solvers (*stifbs*, a Bulirsch-Stoer semi-implicit method with adaptive stepsize control [77], and LSODA, a standard BDF solver with automatic detection of stiffness [75]).

Table 3.1: Execution times (sec) for stiff and non-stiff solvers (rel. error = 10^{-5})

Graph	ODE Solver		
	rkqs [77]	stifbs [77]	LSODA [71]
star: six 2-chains connected to a central vertex	16.51	1.15	1.06
chain10: a 10-chain	11.65	1.28	1.06
dodeca2: two dodecahedra connected by an edge	1393.6	38.16	18.46

3.6 Some Examples

A layout module using the force-directed algorithm described above was implemented in C++ using the class library LEDA [43, 78] and the differential equation solver LSODA [75]. LSODA detects stiffness in the equations as the solution proceeds and automatically switches between stiff and non-stiff methods when appropriate [79]. A logarithmic law was used for springs of type I, and an inverse square law was used for springs of type II as proposed by Eades[20].

The Inventor [80] and LEDA [43] C++ class libraries and a software testbench [81] developed at the University of Newcastle were used to display the final layouts.

Some examples are shown in Figures 3.11, 3.12, 3.13, 3.14, 3.16, and 3.17. Table 3.2 gives execution times for the various algorithms on a Sun SPARC10 with the relative position error parameter set to 0.001. It is unfortunately apparent that even with an efficient integration algorithm, the execution times for even moderately sized graphs can exceed the efficiency requirement for interactive use (i.e. less than two seconds [11]).

Table 3.2: Execution times for the example graphs (rel. error = 10^{-3})

Graph	Figure	Time (sec)	Notes
dodeca2	3.11	10.8	Two dodecahedra connected by an edge
möbius5	3.12	303	Möbius strip with connected satellites
tree	3.13	.45	Tree constrained to a sphere
torus	3.14	20.8	Connected loops drawn on a torus
pentweb	3.16	1.6	Vertices constrained to two spheres
treeweb	3.17	60	Vertices on a cone and sphere

The LSODA software reports that stiff solution methods are needed near the equilibrium configuration for all graphs in this sample, and in fact for most graphs tested. For graphs which are not two–connected, LSODA automatically switches to stiff methods almost immediately.

In the course of this work it was found that the occurrence of stiffness is not uncommon in the mechanical systems arising from applying the spring algorithm to the graph layout problem. If accuracy in establishing the equilibrium configuration is required, as it may be to apply some surface constraints and to avoid distortions of symmetry, stiff solution methods are often significantly more efficient.

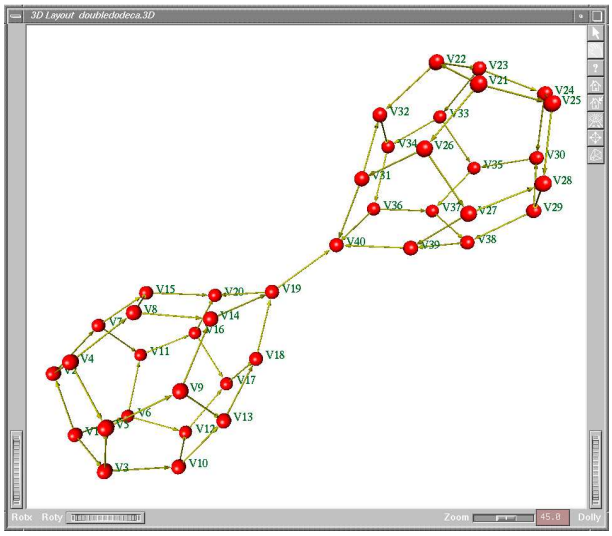


Figure 3.11: 3D spring layout of two dodecahedra connected by a single edge. The distortion is caused by the cumulative repulsive forces between the two dodecahedral components.

3.7 Time Complexity

For the simple methods of ODE integration (e.g. the Euler method), estimates of the maximum computational effort needed are difficult to make, as the total number of iterations to reach an equilibrium depends strongly on the stepsize and system stiffness. However the solvers incorporated in the LSODA software appear in practice to require only a fairly constant number of iterations (i.e. within a factor of about 2), substantially independent of the nature of the input graph. The time for each iteration of course depends on the number of vertices (and edges if sparse–matrix solvers had been used) in the input graph.

Experimentally, the time complexity is somewhere between $O(N^2)$ and $O(N^3)$ depending on the graph. In general, the exponent of N is higher for ODE systems which are inherently stiffer, for example graphs which have a large number of components which are not two–connected.

This is consistent with a simple estimate of the time complexity assuming that a constant number of iterations is required to reach an equilibrium configuration. The cost of one iterative step for explicit ODE solvers is $O(N^2)$ as the solution of an $N \times N$ linear system of equations is required. For systems which are so stiff that they require computation of the Jacobian matrix at each iteration, an additional

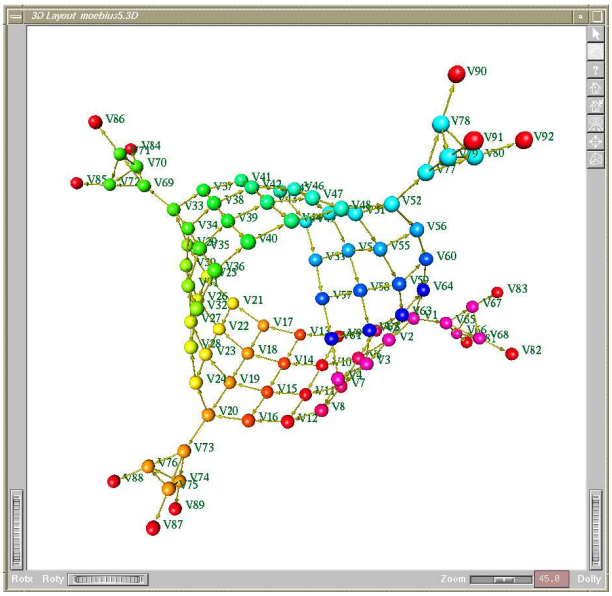


Figure 3.12: 3D spring layout of a Möbius strip with attached satellites.

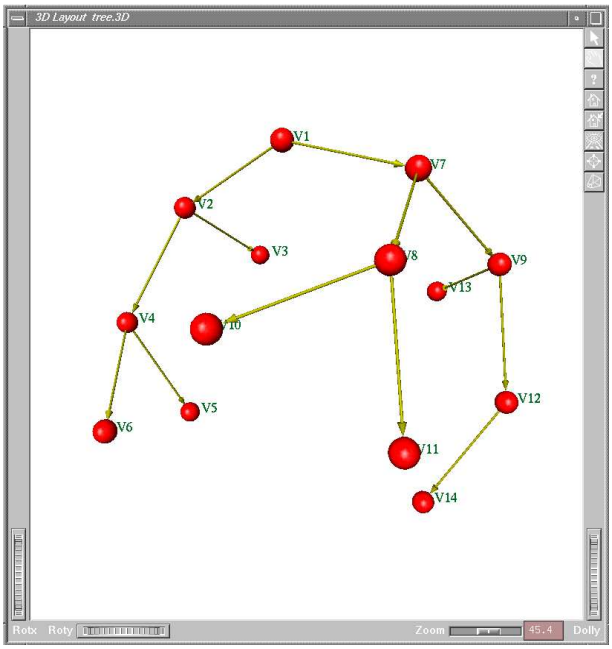


Figure 3.13: 3D spring layout of a tree constrained to lie on a sphere.

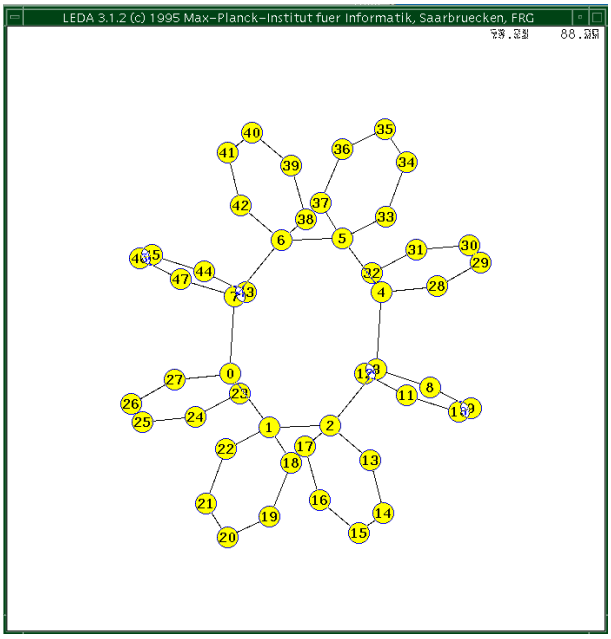


Figure 3.14: 3D spring layout with vertices constrained to lie on a torus.

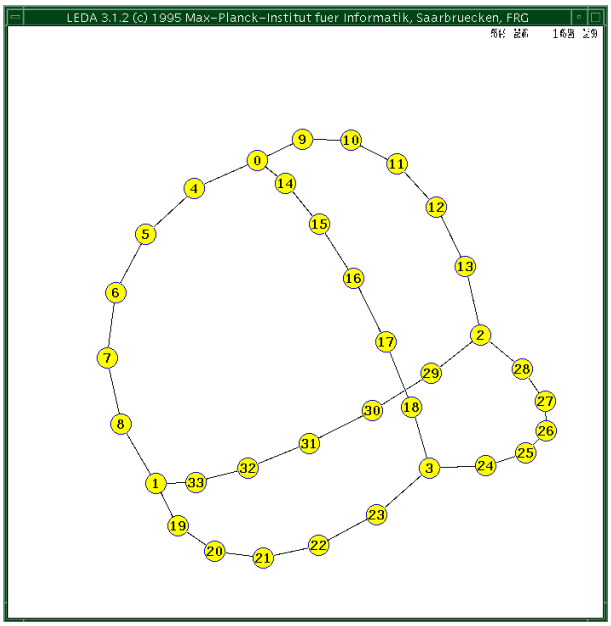


Figure 3.15: 3D spring layout of K_4 (with vertices labeled 0, 1, 2, 3) on a sphere. The edges are approximately constrained to the surface by subdivision.

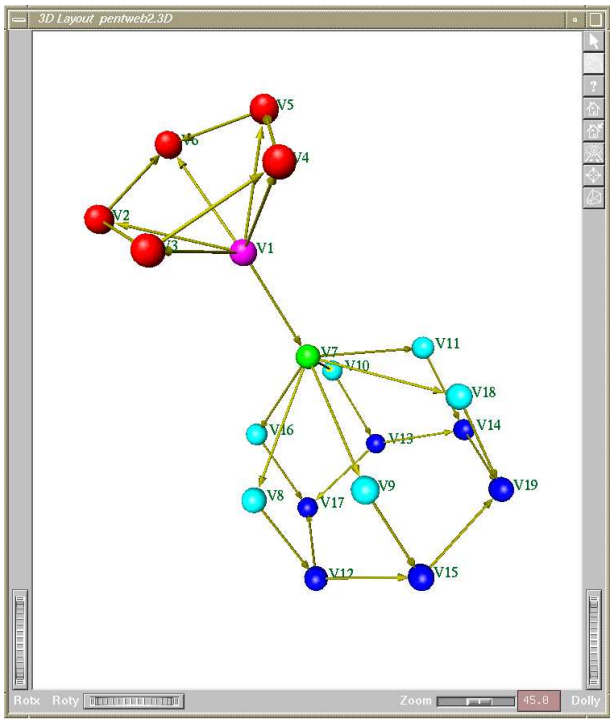


Figure 3.16: 3D spring layout on two spheres, with vertices { V1 – V6 } on one sphere and { V7 – V19 } on the other.

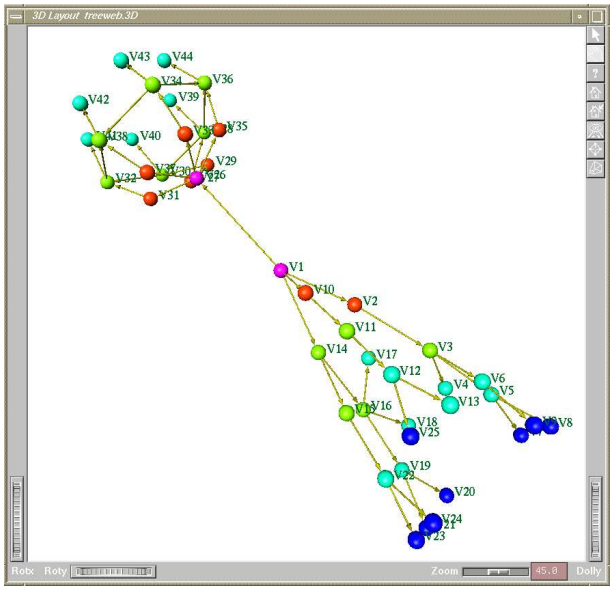


Figure 3.17: 3D spring layout on a sphere and cone.

time of order $O(N^3)$ is required at each iteration, giving an overall time complexity of order $O(N^3)$. In most cases, the Jacobian only has to be updated sporadically, giving an expected time complexity between $O(N^2)$ and $O(N^3)$, consistent with the experimental observations.

3.8 Remarks

Stiff solution methods, while considerably more computationally expensive per integration step, require far fewer steps to reach an equilibrium configuration of a stiff system, giving a significant overall reduction in execution time. As noted above, the execution time of LSODA for medium and larger sized graphs can nevertheless greatly exceed the two seconds limit needed for interactive use.

The LSODA software switches to stiff solution methods when it detects stiffness in the system of differential equations. The same numerical algorithm is applied to all equations in the system. There may be benefits in developing an approach which only applies stiff solution methods to those equations of the system which exhibit stiff behaviour.

As noted by other authors [45], developments in the numerical solution of astronomical N–body problems (also very computationally–intensive, large systems) are a valuable resource for efficient force–directed graph layout. A further improvement in execution speed may be possible by applying an approach in which individual time step sizes are assigned to each equation, based on a local solution accuracy criterion [82, 83].

Chapter 4

3D Layered drawings of directed graphs

Directed graphs are graphs in which the vertices at the ends of each edge form an ordered pair, that is, a direction is associated with each edge. In directed graphs which model relational information, this direction attribute of edges commonly represents a hierarchy corresponding to a time sequence, precedence relation or flow. A good layout algorithm will display this additional information in an easily understandable way.

Sugiyama [18] introduced an effective layout style for 2D drawings of directed graphs. Vertices are positioned on the page in a way which tends to aligns the edge directions so that the hierarchical relationship of the vertices is displayed. Figure 4.1 illustrates the general idea.

The main features of the basic 2D algorithm will be outlined in the next section. Some simple extensions which adapt it to 3D display will be described in Section 4.2. Most stages of the 2D algorithm have direct equivalents in the 3D extensions. More emphasis will be placed in Section 4.1 on those parts of the 2D algorithm which must be modified in the 3D version.

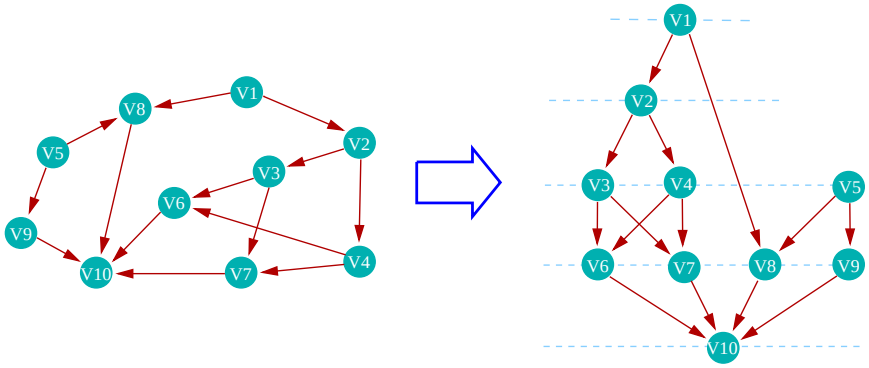


Figure 4.1: An example of the Sugiyama hierarchical drawing style for directed graphs.

4.1 Layered drawings of directed graphs

Layout algorithms based on layering are known to perform well for 2D drawings of directed graphs [18, 84, 85]. The basic approach is to partition the graph into vertex-disjoint sets which are assigned to distinct layers in the drawing. The vertex partition is chosen so that all the edges in the drawing have the same general orientation (e.g. downwards) when the layers are arranged as parallel equi–spaced lines on the page. The algorithm implementing this drawing style proceeds in four stages, roughly corresponding to a sequence of four aesthetic constraints [19]:

1. edges should be oriented in the same general direction,
2. vertices should be assigned to layers corresponding to parallel, equi–spaced lines,
3. edge crossings should be avoided, and
4. edges should be as straight as possible.

The possible coordinates for each vertex are successively restricted at each stage until the last stage completely determines the drawing.

4.1.1 Stage 1: Make the graph acyclic

Clearly, a graph containing cycles cannot be drawn so that all edges point in the same general direction: at least one must point in the reverse direction. The task of the first stage of the algorithm is to identify a set of edges whose reversal

would make the graph acyclic. This set should be as small as possible so that when the edge directions are restored in the final drawing, the visual impression of uni–directional flow is not unduly disturbed.

The problem of finding this set, known as the "minimum cardinality feedback edge set" is NP–hard [41]. Several heuristic algorithms for this problem are reviewed in [19]. The simplest, and least effective, is based on a depth-first search. A root vertex is chosen and vertices are labelled in the order visited by the search. The set of vertex labels which decrease in the edge direction form a feedback edge set.

Although other heuristics known for this problem [86] have significantly superior performance in terms of minimising the size of the feedback edge set, none have attractive performance bounds for the case of general undirected graphs.

4.1.2 Stage 2: Assign vertices to layers

The aim of this stage is to distribute the vertices among an ordered sequence of subsets. Each subset corresponds to a layer in the drawing and will ultimately lie along a single line (perpendicular to the general flow direction).

The vertices of an acyclic directed graph can always be partitioned (usually in many ways) into an ordered sequence of subsets in such a way that the edges have directions consistent with the subset ordering. This permits a drawing with all edges oriented in the same general direction.

Some applications have a natural layering of vertices, for example when a common time is assigned to each subset of a vertex partition. In general however, the layering stage is free to partition the graph in any way consistent with the first aesthetic constraint, and the drawing area available.

The most commonly applied layering algorithm is "longest path layering". An acyclic graph has at least one "sink" vertex having no outwardly-directed edges. These vertices are placed in the lowest layer. The remaining vertices are assigned to higher layers based on their longest path (in the number of edges traversed) to a sink vertex in the lowest layer. Although simple to implement, this algorithm does not attempt to use the available drawing area efficiently. In particular, while the algorithm produces a partition with the minimum number of layers, the individual layers may contain too many vertices for display in a drawing of specified width.

The problem of finding a layering partition which satisfies the first aesthetic constraint with bounds on the drawing dimensions is NP–hard. However, a heuristic which gives drawings with bounded width can be derived from the multiprocessor scheduling problem [19].

Another important related consideration in 2D layered drawings is the occurrence of "long edges", i.e. edges which traverse one or more layers. Long edges are undesirable. It is more difficult to follow long edges visually in the drawing (cf the spring algorithm aesthetic: "all edges should be drawn with the same length"). Furthermore, long edges are usually dealt with by introducing dummy vertices (removed in the final drawing) in the traversed layers so that all edges can be treated together in the next stage of the algorithm. This increases the number of vertices to be positioned, at a corresponding cost in execution time.

The bends in long edges at the dummy vertex positions also make the edges visually more difficult to follow. An additional drawing constraint, that of making long edges as straight as possible, must be incorporated into the algorithm for final placement of vertices in the drawing.

The best strategy appears to be to minimise the number of dummy nodes introduced at the layering stage. Gansner *et al* [87] described a layering algorithm which achieves this.

4.1.3 Stage 3: Minimise the number of edge crossings

The number of edge crossings depends only on the ordering of the vertices (and dummy vertices) in each layer. Unfortunately, determining this optimum ordering is NP–hard [88], even for two layers [89]. Most heuristic algorithms which have been proposed to find approximate solutions fall into two categories. The most common approach is to place the vertices at some form of average of their adjacent vertices' positions. The vertex ordering is then implicit in the ordering of coordinate values. Any collisions which occur must be explicitly recognised and resolved.

There are three major variants of the adjacent-vertex approach. The *barycentric* ("centre of mass") method operates on pairs of adjacent layers. The vertices in one layer L_0 are fixed in position and each vertex in the other layer L_1 is placed at the average position of its adjacent vertices in L_0:

$$x_k = \left(\frac{1}{|N_k|} \sum_{j \in N_k} x_j \right) \equiv \bar{x}_j, \quad j \in N_k,$$

where

x_k is the x-coordinate of the vertex v_k being placed in position,

N_k is the set of indices of the vertices which are adjacent to v_k in L_0, and

x_j is the coordinate of vertex v_j, a vertex in L_0 adjacent to v_k.

The *median* heuristic substitutes the median operation for the averaging operation above. It has been shown that use of the median heuristic results in at most three times the minimum possible number of edge crossings [19].

In both the barycentric and median algorithms, one layer is arbitrarily ordered and given x-coordinates in the initial step. Layers are then considered in a sequential pairwise fashion until all layers have been ordered.

The third variant, known as the *degree–weighted barycentric* ("dwb") heuristic is inspired by an algorithm due to Tutte [90]. The vertices in the two outermost layers are given arbitrary positions. The vertices in the intermediate layers are positioned according to

$$x_k = \frac{1}{2} \left[\frac{1}{|N_k^+|} \sum_{j \in N_k^+} x_j + \frac{1}{|N_k^-|} \sum_{j \in N_k^-} x_j \right] \tag{4.1}$$

where

x_k is the x-coordinate of the vertex v_k being placed in position,

N_k^+ is the set of indices of v_k's inset,

N_k^- is the set of indices of v_k's outset, and

x_j is the x-coordinate of v_j, a vertex adjacent to v_k.

This gives a linear system of equations which can be solved for the x-coordinates of the free vertices. The system can be solved with standard methods. A simple iterative (Gauss-Siedel) algorithm is also effective in practice: first position the free vertices with the barycentric method, and then apply equation 4.1 iteratively to

consecutive layers boustrophedonically until there is no further significant change in the vertex coordinates.

It is known that no heuristic which places vertices on the basis of the positions of its adjacent vertices has a performance bound significantly better than the median heuristic's. In practice, hybrid methods appear to perform best. For example, the vertices might be ordered by the median heuristic and then vertex swapping might be performed until there is no further improvement in the number of edge crossings.

4.1.4 Stage 4: Position the vertices

This stage assigns final coordinates to the vertices in each layer. Discussions describing this stage in the literature generally focus on finding positions for the dummy vertices needed to define long edges. The dummy vertices should be positioned so that bends in the long edges are reduced as much as possible. This is usually expressed as a constrained least-squares problem for which standard numerical methods are available.

If an averaging algorithm has been used in Stage 3 (ordering the vertices) the coordinates have already been assigned to the vertices. However implementations which employ a combinatorial approach in Stage 3 have not assigned coordinate values and a similar constrained optimisation approach as described for the dummy vertices can be formulated to place vertices so that paths (i.e. consecutive edges) are as straight as possible.

Two alternative strategies for positioning vertices are to place them at integer coordinates, thus ensuring a minimum separation, as in common in tree-drawing algorithms, or to use a constrained spring algorithm. In the latter approach, a mechanical model of the drawing is defined in which the vertices (and dummy vertices) are free to move along the lines corresponding to the layers.

An attractive force is applied between vertices connected by an edge and a repulsive force is applied between vertices in the same layer. In this application, the attractive force need not become repulsive at some range corresponding to the desired edge length as the layer spacing guarantees that vertices in different layers cannot move closer together than the layer spacing. This approach will be discussed further in the next section as part of a hybrid algorithm for 3D vertex

placement.

4.2 Extension to 3D layered drawings

The success of the hierarchical layering approach to directed graph layout in 2D suggests that the essential character of the drawing style should be retained in its extension to 3D. Dodson [48] has reported a general approach which constrains vertices to 2D planes corresponding to the layers. This section will describe some 3D layout algorithms which attempt to satisfy the additional 3D aesthetic constraint: vertices should be placed on surfaces. This is motivated by the conjecture that distribution of vertices on surfaces are easier to comprehend in 3D display than distributions throughout a volume. By placing vertices on simple surfaces, the perceptual problems caused by apparent edge and vertex overlap may be reduced.

Most of the stages in the 3D algorithms are identical with their 2D counterpart. In particular, as in the 2D case, the graph must be made acyclic (Stage 1), and the vertices must be assigned to layers (Stage 2). However, the minimisation of the number of edge crossings (Stage 3) is less significant in 3D for two reasons. Firstly, actual edge crossings are less likely to occur when the vertices are distributed in 3D. Secondly, *apparent* edge crossings are in any case inevitable: any graph layout in 3D can be oriented so that two edges appear to cross in at least one point. Long edges, while still incurring a penalty in terms of diagram understandability are also less significant for the same reasons.

In the 3D algorithms to be described, Stages 3 and 4 of the 2D algorithms (edge crossing minimisation and vertex placement) are combined. In the first 3D drawing style, vertices assigned to a layer are constrained to lie in a plane and adjacent layers correspond to adjacent parallel planes. In the second style, in addition to being confined to parallel planes, vertices are also constrained to lie on simple surfaces of revolution (cylinders or cones) whose axes are perpendicular to the layer planes. This effectively constrains the vertices to lie on circles equally spaced along the surfaces' axis of rotation.

4.2.1 Plane layering

After Stage 2 in the basic algorithm has been carried out, the vertices have been assigned to a sequence of ordered layers so that the edges can be drawn with roughly the same orientation.

The function of the final stage in the plane-layering algorithm is to determine coordinates for the vertices so that the vertices in a given layer are confined to the corresponding plane while keeping paths (sequences of consecutive edges) as straight as possible.

The average-position algorithms of section 4.1.3 are applicable to this problem with the modification that both x- and y-coordinates are computed (it is assumed that the z-axis is perpendicular to the parallel planes defining the layers). The barycentric method becomes

$$\begin{aligned} x_k &= \left(\frac{1}{|N_k|} \sum_{j \in N_k} x_j \right) \equiv \bar{x}_j, \quad j \in N_k \\ y_k &= \left(\frac{1}{|N_k|} \sum_{j \in N_k} y_j \right) \equiv \bar{y}_j, \quad j \in N_k \end{aligned}$$

where N_k is the set of the indices of the vertices adjacent to vertex v_k.

As in the 2D case, the positions of the vertices in one layer (the "primary layer") are arbitrarily fixed and the algorithm is applied to consecutive pairs of layers. The median and degree-weighted barycentric heuristics can be modified in the same way. Since the equations for the x- and y-coordinates are completely uncoupled, the algorithms can compute the positions in these directions independently, thus requiring twice the computational effort of the 2D case.

Collisions can again occur whenever two or more vertices in a layer have the same set of adjacent vertices, or due to a poor choice of initial vertex positions in the primary layer. Collision, defined as placement of two vertices at a separation less than some prescribed minimum, must be detected (by calculating the separations of vertex pairs in a layer) and resolved by separating the vertices to a prescribed distance or by altering the vertex positions in the primary layer in the case of accidental collisions. This can be difficult in 3D as the direction of separation must be defined as well as the separation distance to avoid introducing further collisions.

A simpler method is to adopt a hybrid approach to this stage of the layout algorithm: vertices are placed initially using an average-position algorithm such as the barycentric heuristic, and the resulting layout is refined by applying a constrained spring algorithm.

By applying a repulsive force of suitable strength between vertices in the same layer, collisions can be avoided. Edges are represented in the model as springs with zero natural length so that the forces are always attractive. The spacing of planes ensures that vertices in different layers cannot be placed closer than the distance between adjacent planes. The *vertex resolution*, i.e. the minimum distance between vertices in the layout, can be modified by adjusting the relative strength of the repulsive force law relative to the edge-modeling force.

The constrained spring algorithm can be applied to the graph as a whole, or in an iterative fashion by fixing all vertices except those in a single layer. The size of the spring system to be integrated is then just the number of vertices in that layer. Successive iterations compute the equilibrium configuration of successive layers, keeping the other layers fixed. Since the time complexity is at worst approximately $O(N^3)$ (see Chapter 3), the total time taken to converge may be reduced.

At equilibrium, the spring forces tend to keep connected vertices vertically aligned, vertices within a layer separated, and paths traversing consecutive layers as straight as possible.

As an example, Figure 4.2 shows a 2D layered drawing of a directed graph and Figure 4.3 shows the same graph with 3D plane layering.

4.2.2 Additional surface constraints

The visual impression that vertices in a 3D layout lie on parallel planes can be strengthened by such rendering techniques as lightly shading the planes. Alternatively, or in addition, vertices can be constrained to also lie on non-plane surfaces intersecting the planes.

Those most appropriate to layered drawings are the cone and cylinder (considered as a degenerate cone), with axis perpendicular to the planes. The vertices in a layer then lie on a circle. Depending to some extent on the number of vertices in each layer, the circular distribution can enhance the impression that the vertices lie on the (implicit) conical surface.

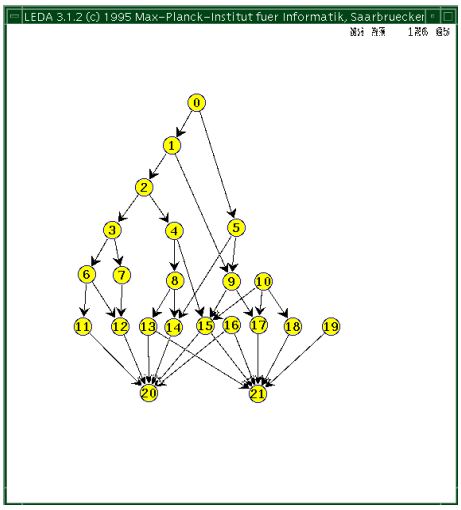


Figure 4.2: Layered drawing of a directed graph.

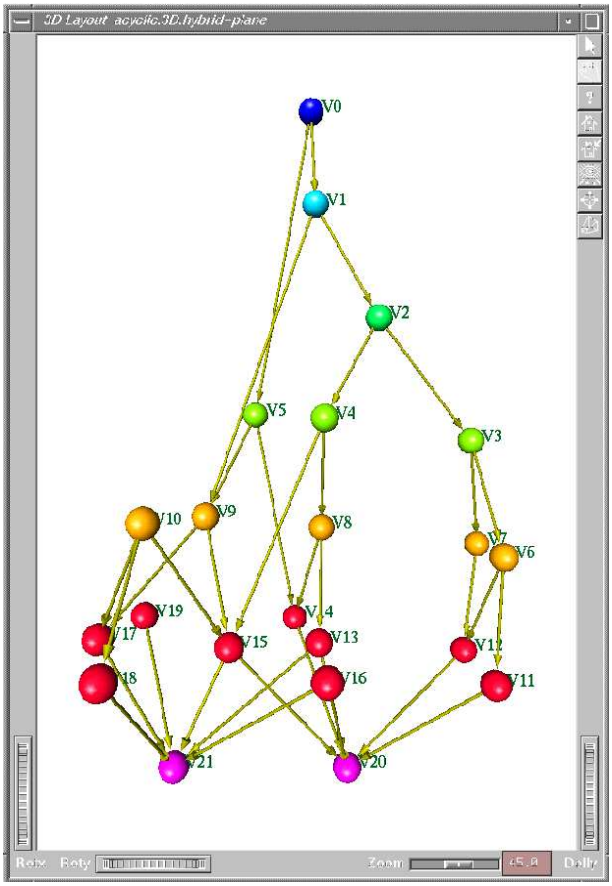


Figure 4.3: Drawing of the graph shown in Figure 4.2 with layers constrained to parallel planes.

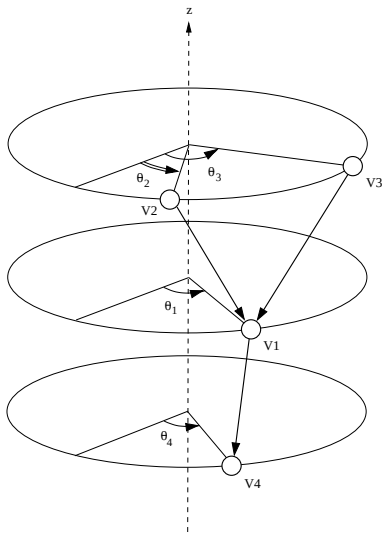


Figure 4.4: Vertex positions are based on the average *angular* coordinates of their adjacent vertices.

Effectively, the drawing has the overall form of a 2D layered drawing wrapped around a cone or cylinder. A reasonable initial choice of cone angle and height can be based on the distribution of vertices into layers. Assuming as a first approximation that the number of vertices in each layer is proportional to the circle circumference of that layer in the surface, an estimate of the minimum dimensions of the development of the cone can be made. This gives a lower bound on the cone base diameter and height.

As in the case of plane layering, the vertex positions can be assigned with one of the average-position approaches discussed above. However, a cylindrical coordinate system (see Figure 4.4) is more appropriate in this case, and position averages are made of the angular coordinates rather than x- and y- coordinates.

Care has to be taken when the angles straddle the radial line $\theta = -180°$. For example, the average of the angles $-170°$ and $+170°$ is $-180°$, not $0°$. A simple way to handle this is to use a complex representation of the vertex positions within the plane corresponding to their assigned layers.

Writing the position of vertex v_j with cartesian coordinates (x_j, y_j, z_j) (z_j being the layer height measured along the cone axis) as

$$\zeta_j = R_j e^{-i\theta_j}$$

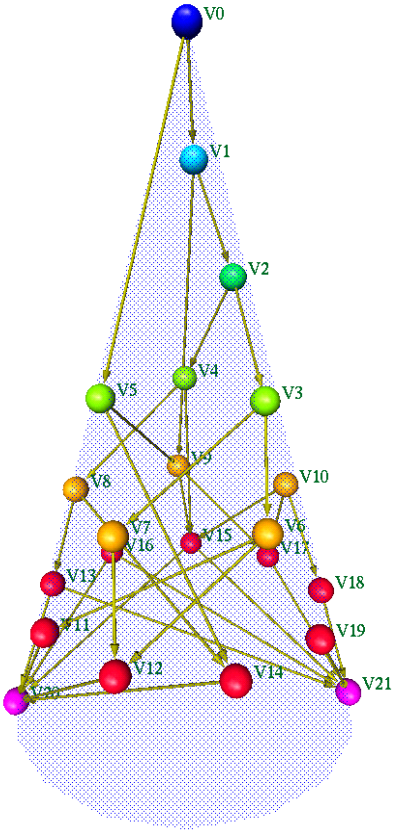


Figure 4.5: Layered drawing on the surface of a cone.

where

$R_j = \sqrt{x_j^2 + y_j^2}$ is the circle radius,

$\theta_j = \arctan(y_j / x_j)$ is the angular coordinate, and

$i = \sqrt{-1}$.

Then

$$x_j = Real\{\zeta\}$$
$$y_j = Imag\{\zeta\}.$$

The average angle $\bar{\theta}$ of a set of vertices $\{\zeta_j\}$, $j = 1, \ldots, M$ is then

$$\bar{\theta} = \arctan \frac{Imag\{\bar{\zeta}\}}{Real\{\bar{\zeta}\}}$$

$$\text{where} \qquad \bar{\zeta} = \frac{1}{M} \sum_{j=1}^{M} \zeta_j .$$

Vertex collisions can occur but can be dealt with in the same ways described previously. Colliding vertices can be separated so that some minimum vertex resolution is achieved, or a refinement of the layout can be made using a constrained spring algorithm applied to the graph as a whole or in an iterated layer-by-layer fashion.

As examples, Figure 4.5 shows a layout of the graph shown earlier in Figure 4.2 on the surface of a cone, made with a hybrid degree-weighted barycentric method followed by a constrained spring refinement.

Figure 4.6 shows a 2D drawing of a directed graph with a different aspect ratio, suggesting that a drawing made on a cylinder would be more appropriate. Figure 4.7 shows this graph drawn with layers constrained to parallel planes, and figure 4.8 shows a layout on a cylindrical surface after applying a hybrid degree-weighted barycentric and constrained spring algorithm. A rotated view of the same layout is shown in Figure 4.9. This end-on view displays the radial-style perspective available with layouts constrained to the cylinder or cone and to some extent illustrates the benefits of being able to vary the viewpoint in 3D.

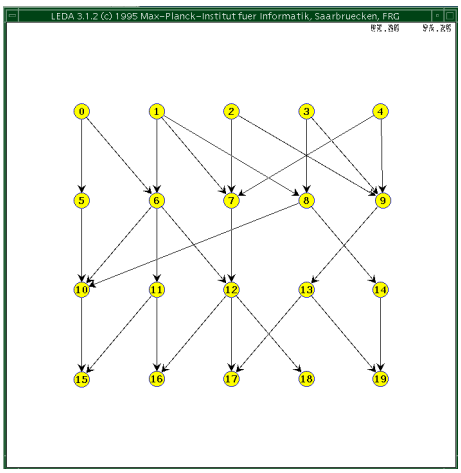


Figure 4.6: Layered drawing of a directed graph with aspect ratio suggesting that a 3D drawing constrained to a cylinder would be appropriate.

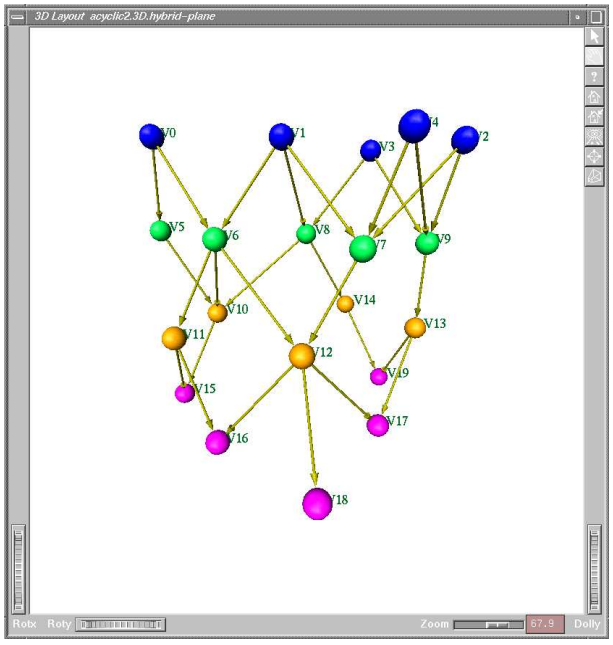


Figure 4.7: Drawing with layers constrained to parallel planes..

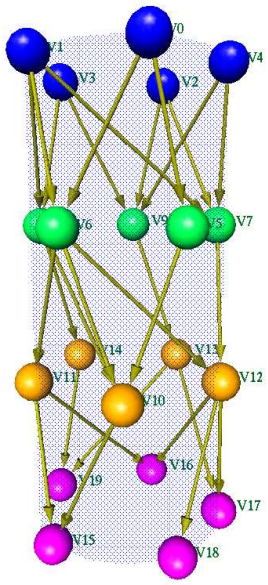


Figure 4.8: Layered drawing on the surface of a cylinder.

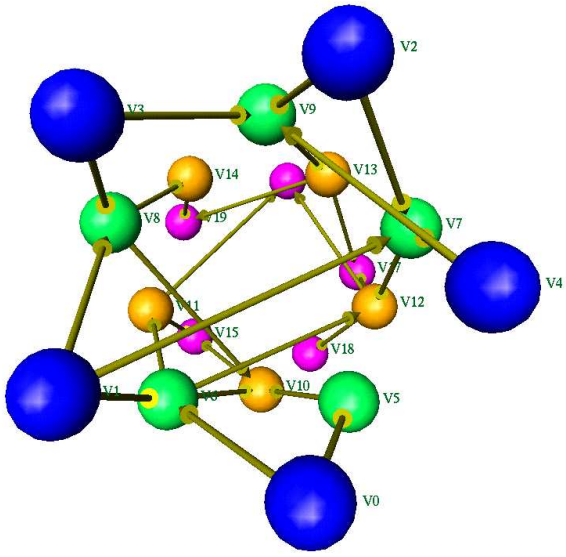


Figure 4.9: Layered drawing on the surface of a cylinder, viewed end – on.

Chapter 5

Concluding Remarks

Graph layout algorithms are an important component of human–computer interfaces designed for the visualisation of relational information. The aim of the present work has been to investigate some practical three–dimensional graph layout algorithms for this purpose. This chapter summarises the principal conclusions drawn in the preceding chapters.

There is a large body of effective methodologies and algorithms for the creation of graph layouts in 2D. Experimental human–factors studies generally suggest that 3D display enhances the readability of graph visualisations. Motivated by this, the algorithms presented here are natural 3D extensions of the force–directed and hierarchical approaches to 2D graph layout.

Current graphics display devices provide only a limited resolution and display area, thus placing a limit on the size of graph which can be displayed effectively. Furthermore, apparent edge–crossings can always be induced in a 3D display by choice of viewpoint. Simplification of the graph before display can reduce these limitations. For example, a partition of the graph into cliques reduces the number of edges and vertices which need to be displayed and also provides a hierarchical way of representing the graph. A heuristic algorithm for partitioning a graph into cliques so that the greatest number of edges can be hidden without loss of information is given in Chapter 2 and is experimentally shown to perform well.

A differential–equation formulation of force–directed methods for undirected graphs is presented in Chapter 3, and it is shown that a large class of drawing

constraints can be incorporated in this framework.

For these algorithms, vertex positions, in the absence of constraints, are essentially determined by the structure of the graph. This is an advantage if the graph has a simple structure, or when the global structure or properties such as symmetries are of particular interest. However, for more typical graphs, force–directed methods distribute the vertices throughout a volume. It is conjectured that the structure of graphs displayed in 3D might be easier to interpret when the vertices are constrained to some suitable 3D surface, suggesting the intuitive 3D drawing aesthetic: *vertices should be placed on simple surfaces.* Surface constraints are readily incorporated in the differential equation formulation and examples of layouts demonstrating the effect of this aesthetic criterion are given.

For interactive use, a layout algorithm should execute quickly. The differential equations describing force–directed methods can exhibit a property known as *stiffness* which requires special–purpose numerical algorithms to achieve the greatest execution speed. Although a significant speed improvement was demonstrated in Chapter 3 through the use of such algorithms, execution speed is still generally inadequate for the prompt interactive display of medium–sized or larger graphs.

The hierarchical approach is known to give an effective 2D drawing of directed graphs. In Chapter 4, this approach was extended to 3D by again constraining the vertices to lie on various simple surfaces. Constraint of the vertices to parallel planes gives the simplest 3D version of the algorithm. Simultaneous constraint to a cone or cylinder perpendicular to the planes confines the vertices to circles in the planes. A hybrid implementation was described in which the vertices are first assigned to layers by a combinatorial algorithm and then geometrically positioned by a surface–constrained force–directed algorithm. These 3D versions of the hierarchical approach retain most of the valuable features of their 2D prototype and also offer a user–selectable viewpoint and a more compact display.

Bibliography

[1] L. Rosenblum et al (ed). *Scientific Visualisation.* IEEE Comp. Soc. Press, 1994.

[2] see for example any issue of the journal IEEE Transactions on Visualization and Computer Graphics.

[3] D. E. Knuth. Computer drawn flowcharts. *Commun. ACM*, 6, 1963.

[4] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-18(1):61–79, 1988.

[5] K. Misue and K. Sugiyama. An overview of diagram based idea organizer: D-abductor. Technical Report IIAS-RR-93-3E, ISIS, Fujitsu Laboratories, 1993.

[6] G. Di Battista, E. Pietrosanti, R. Tamassia, and I. G. Tollis. Automatic layout of PERT diagrams with XPERT. In *Proc. IEEE Workshop on Visual Languages (VL'89)*, pages 171–176, 1989.

[7] C. Williams, J. Rasure, and C. Hansen. The state of the art of visual languages for visualization. In Arie E. Kaufman and Gregory M. Nielson, editors, *Visualization '92*, pages 202–209, 1992.

[8] Wei Lai. *Building Interactive Diagram Applications.* PhD thesis, University of Newcsastle, 1993.

[9] T. Kamada. *Visualizing Abstract Objects and Relations.* World Scientific Series in Computer Science, 1989.

[10] T. Lin and P. Eades. Integration of declarative and algorithmic approaches for layout creation. Technical Report TR–HJ–94–10, Centre for Spatial Information Systems, CSIRO Division of Information Technology, 1994.

[11] B. Schneiderman. *Designing the user interface: strategies for effective human–computer interaction*. Addison–Wesley, 1987.

[12] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994. Currently available from www.cs.brown.edu/people/rt/gd-biblio.html.

[13] see for example Proceedings of Graph Drawing '94 and '95, Springer Lecture Notes in Computer Science series, Vols 894 and 1027.

[14] P. Eades and Petra Mutzel. General techniques for graph drawing. In preparation.

[15] C. Batini, E. Nardelli, M. Talamo, and R. Tamassia. A graph theoretic approach to aesthetic layout of information systems diagrams. In *Proc. 10th Internat. Workshop Graph-Theoret. Concepts Comput. Sci. (Berlin June 198 4)*, pages 9–18, Linz, Austria, 1984. Trauner Verlag.

[16] C. Batini, P. Brunetti, G. Di Battista, P. Naggar, E. Nardelli, G. Richelli, and R. Tamassia. An automatic layout facility and its applications. In *Proc. Internat. Workshop on Software Engineering Environment*, pages 139–157, Beijing, China, 1986. China Academic Publishers.

[17] G. Robins. The ISI grapher: A portable tool for displaying graphs pictorially. Technical Report ISI/RS-87-196, Information Sciences Inst., University of Southern California, 1987. also in Proc. Symboliikka '87 Helsinki Finland August 1987.

[18] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(2):109–125, 1981.

[19] P. Eades and K. Sugiyama. How to draw a directed graph. *Journal of Information Processing*, pages 424–437, 1991.

[20] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[21] Tom Sawyer Software. Graph layout toolkit. Information available from www.tomsawyer.com.

[22] M. Himsolt. Graphed: An interactive graph editor. In *Proc. STACS 89*, volume 349 of *Lecture Notes in Computer Science*, pages 532–533, Berlin, 1989. Springer-Verlag.

[23] C. Batini, P. Brunetti, G. Di Battista, P. Naggar, E. Nardelli, G. Richelli, and R. Tamassia. An automatic layout facility and its applications. In *Proc. Int. Workshop on Software Engineering Environment*, pages 139–157, Beijing, China, 1986. China Academic Publishers.

[24] G. Di Battista, G. Liotta, M. Strani, and F. Vargiu. Diagram server. In *Advanced Visual Interfaces (Proceedings of AVI 92)*, volume 36 of *World Scientific Series in Computer Science*, pages 415–417, 1992.

[25] G. Sander. Graph layout through the VCG tool. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD94, Oct 1994)*, volume 894 of *Lecture Notes in Computer Science*, pages 194–205. Springer, 1995.

[26] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: Animated 3d visualizations of hierarchical information. In *Proc. CHI*, pages 189–193, 1991.

[27] H. Koike. An application of three dimensional visualization to object-oriented programming. In *Advanced Visual Interfaces (Proceedings of AVI '92)*, volume 36 of *World Scientific Series in Computer Science*, pages 180–192, 1992.

[28] S. P. Reiss. 3-D visualization of program information. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pages 12–24. Springer-Verlag, 1995.

[29] R. F. Cohen, P. Eades, T. Lin, and F. Ruskey. Three–dimensional graph drawing. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD94, Oct 1994)*, volume 894 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1995.

[30] Colin Ware, David Hui, and Glenn Franck. Visualising O-O software in three dimensions. In *Proceedings of CASCON '93*, Toronto, Canada, 1993.

[31] Colin Ware and Glenn Franck. Visualising information nets in three dimensions. In *TOG Info. Net. Vis. '94*, Toronto, Canada, August 15 1994.

[32] Helen Purchase, Robert F. Cohen, and Murray I. James. Updating graph drawing aesthetics. In *Proc. GD'95*, volume 1027 of *Lecture Notes in Computer Science*, Berlin, 1996. Springer-Verlag.

[33] F.J. Newbery. Edge concentration: A method for clustering directed graphs. In *Proc. 2nd Int. Workshop on Software Configuration Management*, pages 76–85, 1989.

[34] Q-W. Feng, R. Cohen, and P. Eades. Planarity for clustered graphs. In *Algorithms (Proc. ESA '95)*, volume 979 of *Lecture Notes in Computer Science*, pages 213–226. Springer-Verlag, 1995.

[35] Q-W. Feng, R. Cohen, and P. Eades. How to draw a planar clustered graph. In *Computing and Combinatorics*, volume 959 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, 1995.

[36] Fwu-Shan Shieh and Carolyn L. McCreary. Directed graphs drawing by clan-based decomposition. In *Proc. GD'95*, volume 1027 of *Lecture Notes in Computer Science*, Berlin, 1995. Springer-Verlag.

[37] A. Ehrenfeucht and G. Rozenberg. Theory of 2-structures, part I: Clans, basic subclasses, and morphisms. *Theoretical Computer Science*, 70:277–303, 1990.

[38] A. Ehrenfeucht and G. Rozenberg. Theory of 2-structures, part II: Representation through labeled tree families. *Theoretical Computer Science*, 70:305–342, 1990.

[39] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York and London, 1972. Plenum Press.

[40] J. Bhasker and Tariq Samad. The clique partitioning problem. *Computers Math. Applic.*, 22(6):1–11, 1991.

[41] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, Ca., 1979.

[42] C. J. Tseng. *Automated synthesis of data paths in digital systems.* PhD thesis, Dept of Electrical and Computer Engineering, Carnegie-Mellon University, 1984.

[43] S. Naher and Kurt Melhorn. LEDA, a library of efficient data types and algorithms. In *Proceedings of the 14th Symposium on Mathematical Foundations of Computer Science*, volume 379 of Lecture Notes in Comuter Science, pages 88–106. Springer-Verlag Berlin, 1989.

[44] Xuemin Lin and Peter Eades. Spring algorithms and symmetry. To appear.

[45] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experience*, 21(11):1129–1164, 1991. also as Technical Report UIUCDCS-R-90-1609, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1990.

[46] B. Monien, F. Ramme, and H. Salmen. A parallel simulated annealing algorithm for generating 3D layouts of undirected graphs. In *Proc. GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 396–408, Berlin, 1996. Springer-Verlag.

[47] H. Goldstein. *Classical Mechanics.* Addison Wesley, 1980.

[48] D. Dodson. COMAIDE: information visualisation using cooperative 3D program layout. In *Proc. GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 190–201, Berlin, 1996. Springer-Verlag.

[49] K. Sugiyama and K. Misu. A simple and unified method for drawing graphs – magnetic spring algorithm. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD94, Oct 1994)*, volume 894 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 1995.

[50] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[51] J. E. Dennis and R. B. Schnabel. *Numerical methods for unconstrained optimisation and nonlinear equations.* Prentice-Hall, 1983.

[52] T. Kamps and J. Kleinz. Constraint–based spring–model algorithm for graph layout. In *Proc. GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 349–360, Berlin, 1996. Springer-Verlag.

[53] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. Technical Report CS89-13, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, July 1989.

[54] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD94, Oct 1994)*, volume 894 of *Lecture Notes in Computer Science*, pages 389–403. Springer, 1995.

[55] Ingo Bruss and Arne Frick. Fast interactive 3-D graph visualisation. In *Proc. GD'95*, volume 1027 of *Lecture Notes in Computer Science*, Berlin, 1995. Springer-Verlag.

[56] X. Mendonca. *Heuristics for Planarization by Vertex Splitting*. PhD thesis, University of Newcastle, 1992.

[57] Isabel F. Cruz and Joseph P. Twarog. 3D graph drawing with simulated annealing. In *Proc. GD'95*, volume 1027 of *Lecture Notes in Computer Science*, Berlin, 1996. Springer-Verlag.

[58] C. Kosak and J. Marks. A parallel genetic algorithm for network-diagram layout. In *Proc. 4th Int. Conf. on Genetic Algorithms (ICGA91)*, 1991.

[59] S. R. Mangano. Algorithms for directed graphs. *Dr. Dobb's Journal*, pages 92–97, April 1994.

[60] Rayleigh. *The Theory of Sound, Vol 1.* Dover, 2nd edition, 1945.

[61] M. W. Hirsch and S. Smale. *Differential Equations, Dynamical Systems and Linear Algebra.* Academic Press, 1974.

[62] R. Tamassia and I. G. Tollis. Representations of graphs on a cylinder. *SIAM J. Disc. Math.*, 4(1):139–149, February 1991.

[63] S. Mehdi Hashemi, A. Kisielewicz, and I. Rival. Upward drawings on planes and spheres. In *Proc. GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 1996.

[64] J. Kratochvil and T. Przytycka. Grid intersection and box intersection graphs on surfaces. In *Proc. GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 365–372, Berlin, 1996. Springer-Verlag.

[65] P.Garvan and P.Eades. Drawing genus–1 graphs on the torus. Technical Report TR 96-05, Department of Computer Science and Software Engineering, University of Newcastle, 1996.

[66] D. Baraff. Interactive simulation of solid bodies. *IEEE Computer Graphics and Applications*, pages 63–75, May 1995.

[67] V. I. Arnol'd. *Mathematical Methods of Classical Mechanics*. Springer, 1978.

[68] L. F. Shampine. *Numerical Solution of Ordinary Differential Equations*. Chapman and Hall, 1994.

[69] S. O. Fatunla. *Numerical Methods for Initial Value Problems in Ordinary Differential Equations*. Academic Press, 1988.

[70] C. W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice–Hall, 1971.

[71] Alan C. Hindmarsh and Linda R. Petzold. Algorithms and software for ordinary differential equations and differential-algebraic equations, part I. *Computers in Physics*, 9(1):34–41, Jan/Feb 1995.

[72] Alan C. Hindmarsh and Linda R. Petzold. Algorithms and software for ordinary differential equations and differential-algebraic equations, part II. *Computers in Physics*, 9(2):148–155, Mar/Apr 1995.

[73] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II*. Springer-Verlag, 1991.

[74] C. F. Curtiss and J. O. Hirschfelder. Integration of stiff equations. *Proc. Nat. Acad. Sci. U.S.A.*, 38:235, 1952.

[75] Alan C. Hindmarsh. ODEPACK, a systematised collection of ODE solvers. In R. S. Stepleman et al, editor, *Scientific Computing*, page 55. North-Holland Amsterdam, 1983.

[76] A. C. Hearn. *REDUCE User's manual.* The Rand Corporation, Santa Monica (CA), 1991.

[77] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 2nd edition, 1992.

[78] S. Naher and Kurt Melhorn. LEDA, a platform for combinatorial and geometric computing. To appear.

[79] Linda R. Petzold. Automatic selection of methods for solving stiff and non-stiff systems of ordinary differential equations. *SIAM J. Sci. Stat. Comput.*, 4(1):136–148, March 1983.

[80] Josie Wernecke. *The Inventor Mentor: Programming Object-oriented 3D Graphics with Open Inventor, release 2.* Addison Wesley, 1994.

[81] Robert F. Cohen, Dean Fogarty, Paul Murphy, and Diethelm Ostry. Animated three-dimensional information visualisation. *Australian Computer Science Communications (Proc ACSC'96)*, 18(1):409–416, 1996.

[82] Junichiro Makino. Optimal order and time-step criterion for Aarseth-type N-body integrators. *Astrophysical Journal*, 369:200–212, 1991 March 1.

[83] Sverre J. Aarseth. Direct Methods for N-Body Simulations. In *Multiple Time Scales*, pages 377–418. Academic Press, Inc., 1985.

[84] K. Sugiyama. A cognitive approach for graph drawing. *Cybernetics and Systems: An International Journal*, 18:447–488, 1987.

[85] E.R. Gansner, E. Koutsofios, S.C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, March 1993.

[86] P. Eades, X. Lin, and R. Tamassia. A new approach for drawing a hierarchical graph. In *Extended abstract in Proceedings of the Second Canadian Conference on Computational Geometry*, pages 143–146, 1990.

[87] E.R. Gansner, S.C. North, and K.P. Vo. Dag – a program that draws directed graphs. *Software – Practice and Experience*, 18(11):1047–1062, 1988.

[88] M.R. Garey and D.S. Johnson. Crossing number is NP–complete. *SIAM J. Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[89] P. Eades, B. McKay, and N. Wormald. On an edge crossing problem. In *Proc. 9th Australian Computer Science Conference*, pages 327–334, 1986.

[90] W.T. Tutte. How to draw a graph. *Proceedings London Mathematical Society*, 3(13):743–768, 1963.