A LAYOUT ALGORITHM FOR UNDIRECTED COMPOUND GRAPHS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Erhan Giral

August, 2005

ABSTRACT

A LAYOUT ALGORITHM FOR UNDIRECTED COMPOUND GRAPHS

Erhan Giral
M.S. in Computer Engineering
Supervisor: Assist. Prof. Dr. Uğur Doğrusöz
August, 2005

Graph layout is an important problem in information visualization. All data-driven graph-based information visualization systems require some sort of an automatic geometry generation mechanism, as it is generally not directly available from the data being modeled. This is why graph layout problem has been studied extensively. However, for the case of compound graphs, there are still important gaps in this area. We present a new, elegant algorithm for undirected compound graph layout. The algorithm is based on the traditional force-directed layout scheme with extensions to handle nesting, varying node sizes, and possibly other application-specific constraints. Experimental results show that the execution time and quality of the produced drawings with respect to commonly accepted layout criteria are quite satisfactory. The algorithm has also been successfully implemented as part of a pathway integration and analysis toolkit named PATIKA for drawing complicated biological pathways with compartmental constraints and arbitrary nesting relations to represent molecular complexes and various types of pathway abstractions.

Keywords: Visualization, Graph Visualization, Graph Drawing, Force Directed Graph Layout, Compound Graphs.

ÖZET

YÖNSÜZ BİLEŞİK ÇİZGELER İÇİN YERLEŞİM ALGORİTMASI

Erhan Giral
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yard. Doç. Dr. Uğur Doğrusöz
Ağustos, 2005

Çizge yerleşimi bilgi görselleme alanındaki önemli bir problemdir. Bütün veriye dayalı çizge tabanlı bilgi görselleme sistemleri bir özdevimli geometri yaratma düzeneğine ihtiyaç duymaktadır. Çünkü geometri bilgisi çoğunlukla modellenen bilgide bulunmaz. Bu nedenle çizge yerleşim probleminin etraflıca incelenmesine rağmen bileşik çizgeler durumu aynı kapsamda araştırılmamıştır. Bu çalışmada yönsüz bileşik çizgeler için yeni bir yerleşim algoritması sunulmaktadır. Algoritma, geleneksel güce-dayalı yerleşim şablonunu esas almakta ve iç içelik, değişebilir düğüm şekli ve muhtemel diğer uygulamaya özel kısıtları halledebilecek şekilde geliştirmektedir. Deneysel sonuçlar hesaplama zamanı ve genelde kabul edilen yerleşim niteliği açısından algoritmanın son derece başarılı olduğunu ortaya koymaktadır. Algoritma, bir yolak bütünleştirme ve analiz araç takımı olan Patika için de başarılı bir şekilde gerçekleştirilmiştir. Patika son derece karmaşık yolak bilgisini görsellemektedir ve birçok değişik çeşit biyolojik yolağı görselleyebilmek için, alansal kısıtlar ve rastgele iç içelik ilişkileri içermektedir.

Anahtar sözcükler: Görselleme, Çizge Görselleme, Çizge Çizimi, Çizge Yerleşimi, Güce-dayalı Çizge Yerleşimi, Bileşik Çizgeler.

Acknowledgement

I would like to express my gratitude to my supervisor Assist. Prof. Dr. Uğur Doğrusöz for his efforts in the supervision of the thesis.

I would like to express my special thanks to Assist. Prof. Dr. Uğur Güdükbay and Assoc. Prof. Dr. İsmail H. Toroslu for showing keen interest to the subject matter and accepting to read and review the thesis.

I would like to thank to my family for their guidance and support.

Contents

List of Figures

Chapter 1

Introduction

A graph is an abstract structure that is used to model relational information. Many information visualization systems require graphs to be drawn so that information being modeled becomes human interpretable.

There are various graphical representations for graphs. Usually, vertices are represented by symbols such as points, boxes or ellipses and edges are represented by curves connecting the symbols that represent the associated vertices [11]. However, graphical representations vary greatly according to the application domain. Even within a graphical representation schema, there are infinitely many ways to draw a graph, by simply changing coordinates of nodes in the plane.

When drawing a graph, we would like to take into account a variety of aesthetic criteria. For example, planarity and the display of symmetries are often highly desirable in visualization applications [11]. In general, in order to improve the readability of drawings, it is important to keep the number of crossings and bends low. Also, to avoid wasting of space on screen or page, it is important to keep area of the drawing as small as possible. Trade-offs are often necessary as these are conflicting objectives.

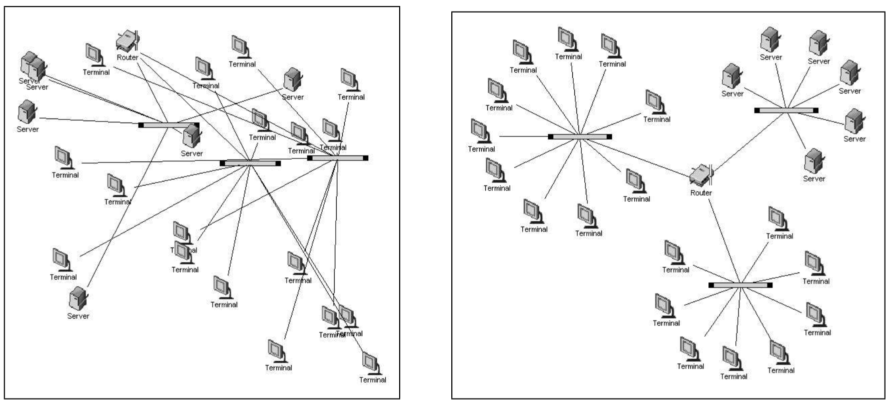

Figure 1.1: Two drawings of a computer network system.

1.1 Aesthetics

Aesthetics is a subjective term, however it is possible to formalize it in our context. An aesthetic property specifies a geometric asset of underlying graph that we would like to highlight as much as possible. Commonly adopted aesthetics are [11]:

- *Crossings:* Minimization of edge-edge crossings is one of the most important aesthetic criteria. Ideally we would like to have crossing free drawings, however non-planar graphs do not admit one. Node-edge crossings should also be minimized, although they are not as important as edge-edge crossings.
- *Overlaps:* Minimization of node-node overlaps is another important aesthetic criterion.
- *Uniform Edge Length:* Minimization of the variance of the lengths of the edges.
- *Symmetry:* Maximize displayed symmetries in the graph.
- *Area:* Minimization of the total area of the drawing. The ability to generate drawings that use screen area efficiently is very important as screen space is an important and generally very limited resource for visualization applications.

Trying to satisfy all of these criteria is generally infeasible if not impossible, as they are inherently conflicting. So one has to prioritize according to needs of a particular application.

1.2 Constraints

Today's demanding visualization systems require more than just aesthetics, in which we establish metrics to define good drawings, universally. Many visualization use cases require us to support several controls that the user can utilize to influence the automated drawing process in some manner. These controls enable users to create drawings that have a desired property that is, users can incorporate their own aesthetic criteria by means of constraints.

Constraints are generally defined for subgraphs [11]. For example in biological pathway drawings substrates of reactions are conventionally grouped together, whereas products of the same reaction are grouped separately and these two groups are drawn as far as possible from each other.

Commonly used constraints in visualization systems are:

- *Fixed location:* Layout algorithm may be restricted to place a subgraph within a certain area (see Figure 1.2).
- *Relative position:* A subgraph would be needed to be placed to a relatively fixed position with respect to another subgraph (see Figure 1.2).
- *Flow:* A directed graph may be placed according to a flow constraint (top-bottom, left-right).
- *Size:* It is possible to allow users to specify size limits for the final drawing.
- *Aspect ratio:* Targeted drawing area is generally constrained by the physical rendering system. It is also possible to allow users supply an aspect ratio for the layout algorithm.

Constraints are very powerful tools for users to have graphs drawn for a particular purpose. However they pose an additional difficulty to layout problem and not all layout algorithms can provide good support for them.

1.3 Use of compound graphs

As graphical user interfaces have improved, and more state-of-the-art software tools have incorporated visual functions, interactive graph editing and diagramming facilities have become important components in visualization systems [12].

Whenever the visualization requirements of the system becomes complex a new scheme for managing complexity is required. In the past years complexity management issue took serious attention. Some studies [17, 22, 31] describe how to extend graphs with a hierarchical structure. Some frameworks were designed to specifically create clusters based on a given data set [26, 15]. HGV [28] is a framework with support for multiple views and hierarchies. Also this issue took serious attention from the industry as well. The systems described in [34, 36, 23, 3, 2, 1], are just a few examples where complexity management is part of a visualization framework or application.

The notion of compound graphs has been used in the past to represent more complex type of relations or varying levels of abstractions in data [24, 27, 21, 13] (see Figures 1.3 and 1.4).

Effective analysis of the underlying data in graph visualization is only possible with sound automatic layout capabilities of such systems. We will continue with background on existing layout algorithms.

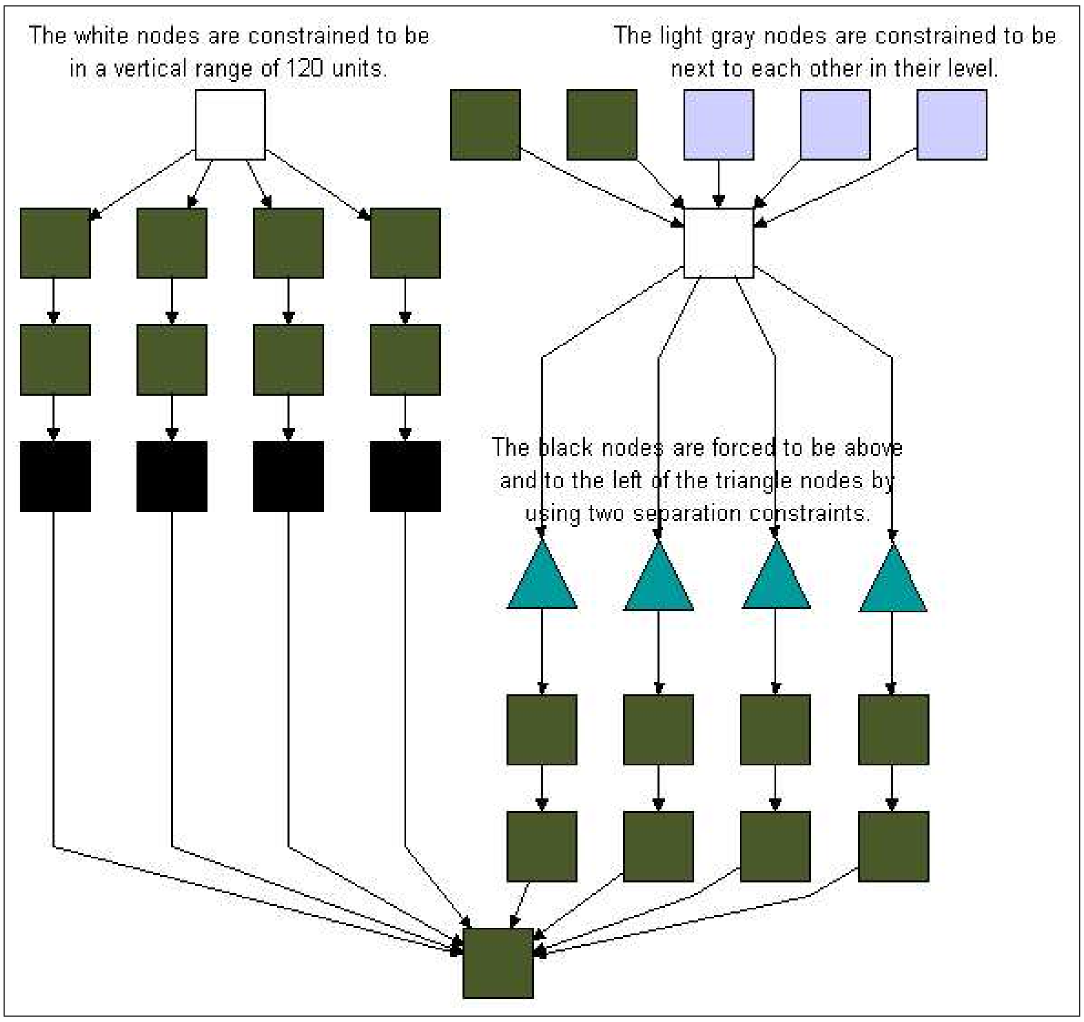
The white nodes are constrained to be in a vertical range of 120 units.
The light gray nodes are constrained to be next to each other in their level.
The black nodes are forced to be above and to the left of the triangle nodes by using two separation constraints.

Figure 1.2: A constraint based layout example. Here we see examples for *fixed location* and *relative position* constraints (courtesy of Tom Sawyer Software).

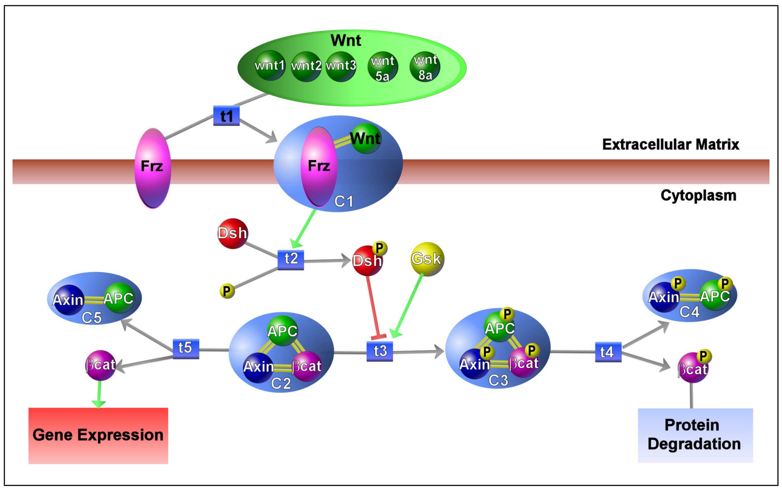

Figure 1.3: Part of a sample compound pathway.

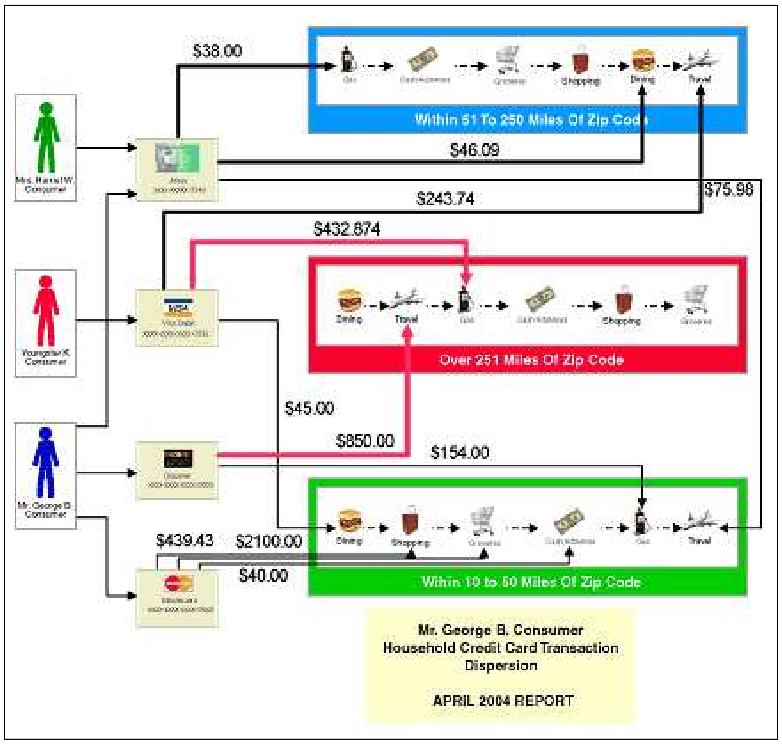
$38.00
Within 51 To 250 Miles Of Zip Code
$46.09
$75.98
$243.74
$432.874
Over 251 Miles Of Zip Code
$45.00
$850.00
$154.00
$439.43
$2100.00
$40.00
Within 10 to 50 Miles Of Zip Code
Mr. George B. Consumer
Household Credit Card Transaction Dispersion
APRIL 2004 REPORT

Figure 1.4: A financial chart (courtesy of Tom Sawyer Software).

Chapter 2

Definitions

A *graph* G is defined by two finite sets V and E, where the elements of V are the *nodes* of G, and the elements of E are the *edges* of G.

A *graph manager* $M = (S, I, F)$ is a structure based on compound graphs, defined by a *graph set* $S = \{G_1, G_2, \ldots, G_l\}$, an *intergraph* I, and a rooted *nesting tree* $F = \left(V^F, E^F\right)$. Each graph $G_i \in S$, each node $v \in V^{G_i}$, and each edge $e \in E^{G_i}$ is represented by a distinct node in V^F. For each node $v \in V^{G_i}$, there exists an edge $(G_i, v) \in E^F$ and for each edge $e \in E^{G_i}$, there exists an edge $(G_i, e) \in E^F$, representing ownership relations in the graph manager. Then G_i is called the *owner* of v (or e); conversely v (or e) is called a *member* of G_i.

A *nesting* of a graph in its parent node facilitates drawing of multiple graphs of a graph manager and their inter-relations simultaneously. The node within which a graph is nested is said to be *expanded*. Expanded node sizes are as big as the boundaries of the associated nested graph. This is represented in the nesting tree by an edge $(n, G_i) \in E^F$ between a node n and a graph G_i, where G_i is not the owner of n. G_i is said to be the *child graph* of the *parent member* m. The graph at the root of the nesting tree is simply called *root graph*.

Another way of associating two different graphs in a graph manager $M = (S, I, F)$ is via the intergraph I. Let $u \in V^{G_i}$ and $v \in V^{G_j}$ be two nodes where

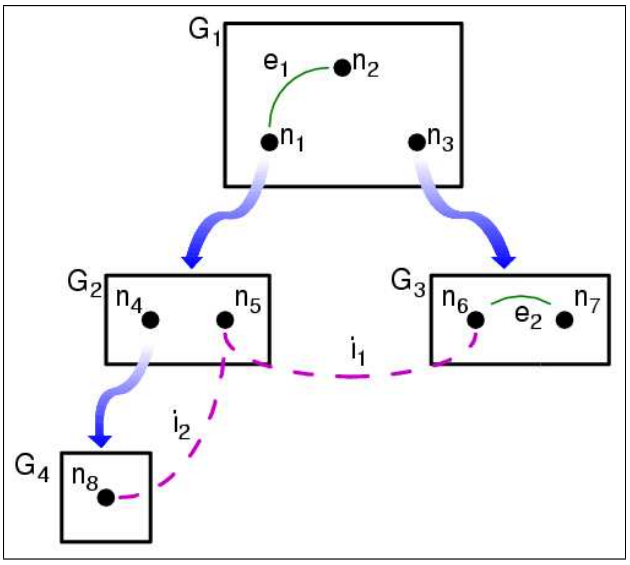

Figure 2.1: A pictorial representation of the graph manager with inclusion tree in Figure 2.2. The gradient arrows show inclusion relations and the dashed edges are the intergraph edges [13].

$G_i \neq G_j$ and $G_i, G_j \in S$. Then the edge $(u, v) \in I$ is called an *intergraph edge*, representing a relation between objects (nodes) that belong to different entities, graphs G_i and G_j in this case.

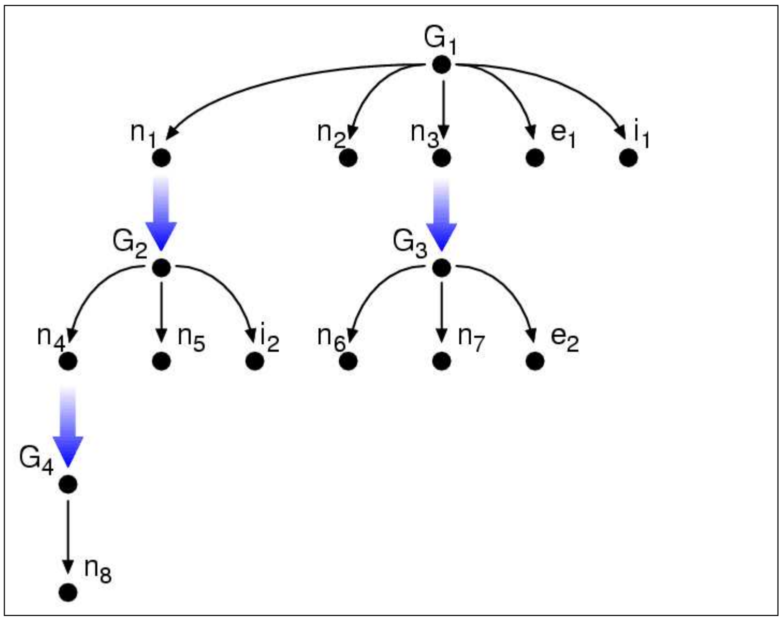

Figure 2.2: The inclusion tree of a graph manager, representing both ownership (solid) and inclusion (gradient) relations [13].

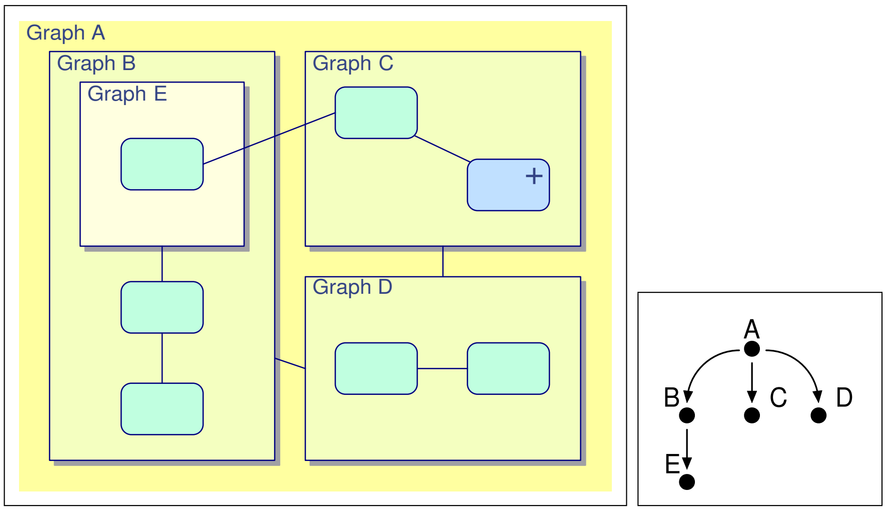

Figure 2.3: Drawing of a graph manager with multiple levels of nesting realized [13].

Chapter 3

Related Work

As we have stated in previous section, good drawings of graphs involve some sort of prioritization of a set of aesthetic criteria. There is no universal algorithm that will generate beautiful drawings for every kind of application-graph. Therefore there are many algorithms in the literature that try to generate good automatic drawings of family of graphs.

Now we will analyze existing work on graph layout. We will first have a look at the state of art in non-compound graph layout and then compound graph layout.

There has been a great deal of work done on general graph layout [11]. However it is possible to classify all of the major algorithms with following classification.

3.1 Orthogonal graph layout

Orthogonal layout places the nodes of the graph on a grid of rows and columns, and routes the edges strictly parallel to the x and y axes. Edges are drawn as multi-lines however these lines are either vertical or horizontal, hence edge-edge crossings are somewhat easier to read. Also as nodes are placed on a grid node spacing is regular by definition However edges are generally quite long, hence not easy to track (see Figure 3.1). It is also possible to integrate incremental layout support into orthogonal layout algorithms [36].

There are different algorithms in the market that try to efficiently generate orthogonal drawings [6, 9, 8].

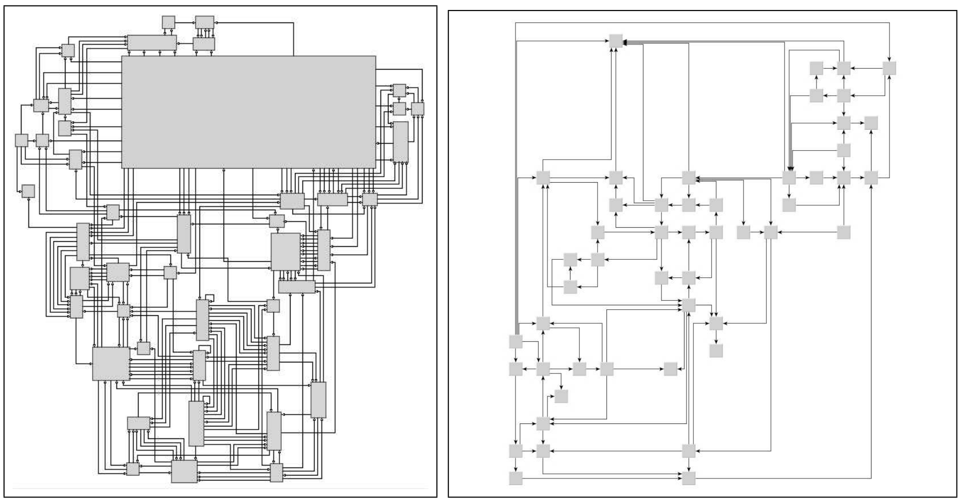
Figure 3.1: Two orthogonal drawings (left: courtesy of Tom Sawyer Software, right: courtesy of Tom Sawyer Software yFiles).

3.2 Hierarchical graph layout

Hierarchical layout algorithms aims to highlight the main direction or flow within a *directed* graph (see Figure 3.3). In hierarchical layout nodes are placed in hierarchically arranged layers. The ordering of the nodes within each layer is chosen in such a way that edge crossings are as small as possible [33, 29].

Edges are drawn as multi-lines however these lines may have arbitrary slopes or even be replaced by curves. Like orthogonal algorithms, hierarchical layout places nodes regularly, which is a great readability advantage. For the time being hierarchical layout has the best support for compound nodes, as we will highlight later.

However if graph has cycles, meaning data being represented is not exactly hierarchic, there will be ridiculously long back edges (see Figure 3.2). Therefore if your data model allows cycles in graphs hierarchical graph layout may not be a right choice.

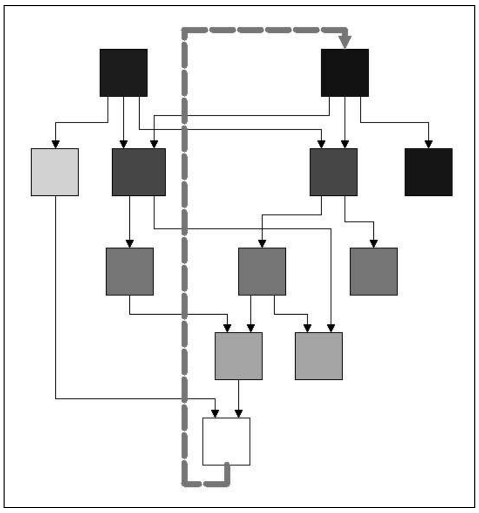

Figure 3.2: A hierarchical drawing of a cyclic graph. Cycles cause very bad back edges in hierarchical drawings. Here such a back edge is depicted as thick-dashed.

Typically all algorithms that fall into this category has following flow:

1. *Partitioning into layers*
2. *Reduction of crossings*
3. *Coordinate Calculation*
4. *Edge Routing*

Hierarchical layout is a good choice to visualize directed graphs as it utilizes inherent hierarchy information in the graph. It is possible to define and maintain constraints easily [34]. Hierarchical graph layout has excellent incremental layout support as well [34, 36, 23].

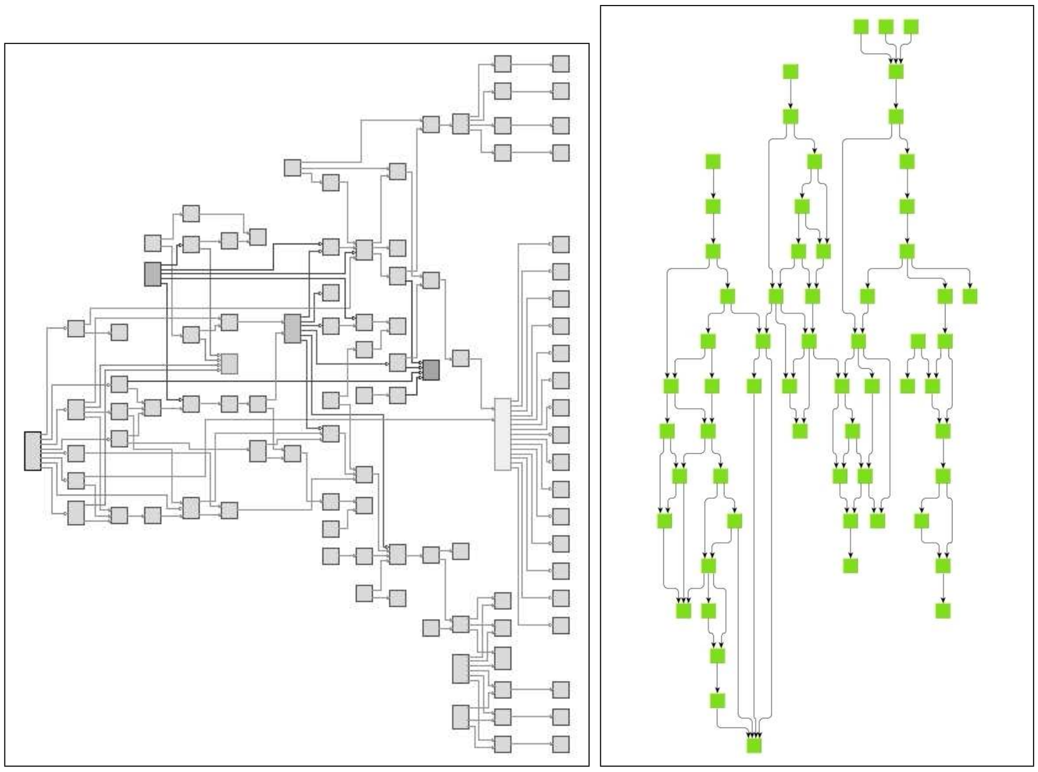

Figure 3.3: Two very good hierarchical drawing samples. (left: courtesy of Tom Sawyer Software, right: courtesy of yFiles)

3.3 Circular graph layout

Circular layout algorithms emphasize group and tree structures within a graph. It gives excellent drawings for graphs that have certain topological properties, that is graphs containing trees and clusters [14].

In circular layout, nodes are first clustered according to topology and each cluster is embedded on a circle in a way that edge crossings within the cluster is minimized. Then clusters are placed in a way that minimizes inter cluster edges (see Figure 3.4).

Having assumptions on topology, renders circular layout algorithm poor against generic graphs.

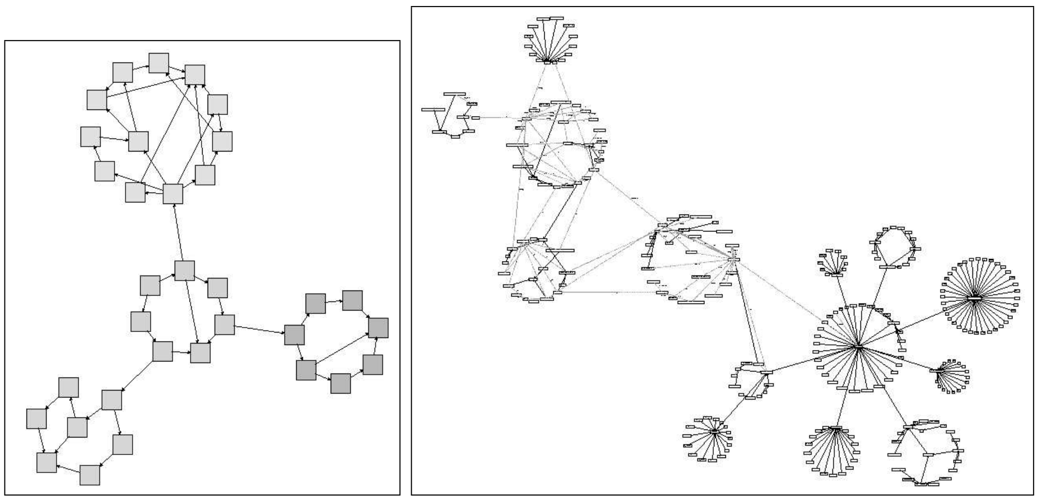

Figure 3.4: Two circular drawing samples. (courtesy of Tom Sawyer Software)

Circular layout is a good choice to visualize trees and highly clustered graphs. State of the art implementations are [34, 36]. However it is hard to support compound nodes as a compound node may span multiple clusters.

3.4 Force directed graph layout

In force directed layout, graph to be laid out is represented as a physical model and a simulation of this model is done with a feasible accuracy. That is graph layout problem is solved via simulating a physical system.

In the basic model, nodes are represented as charged particles that repel each other and edges are represented with springs. The energy level of a node is determined from the forces acting on it. The spring embedder tries to minimize the global energy level by moving the nodes in the direction of the forces. Global energy level, which is the sum of all energy levels of the nodes, is computed after each iteration of the system to determine if the total energy is below a certain amount.

The accuracy and reality of this basic system is a trade off between performance and quality. Generally, for performance reasons only one node is displaced at a time [25].

It is always possible to include additional physical factors in the model to have more realistic hence better resulting systems, like:

- *Magnetic forces:* These are generally used to enforce a flow in to the drawing. For example; in directed graphs it is possible to emphasis the flow if all edges are interpreted as compasses that align themselves according to a magnetic field [32].
- *Gravitational forces:* These are generally used to produce more compact drawings. As spring forces are only effective within components and repulsive forces can make the drawing only bigger. Basic model should be extended to be able to minimize inter component space. Hence gravitational forces are introduced. All nodes are attracted to the mass center of all the other nodes [3].
- *Acceleration:* It is also possible to add mass related factor momentum into the model. This factors adds the previous velocity of a node to the movement being calculated for an iteration. Acceleration is generally added to improve running time performance as well as quality.
- *Temperature:* The basic model with its extensions described until now can settle with a local minima. To overcome this problem it is possible to use controlled amount of randomness. Researchers in optimization theory use a technique from statistical mechanics called *simulated annealing* allowing for changes into states with higher energy. With this addition calculated node movement is disturbed with a relatively minor random vector to avoid being trapped at a local energy minimum. At the beginning the magnitude of this random vector is bigger, as the simulation matures the system is cooled down meaning magnitude of the force is reduced in order to stabilize the final layout [18].

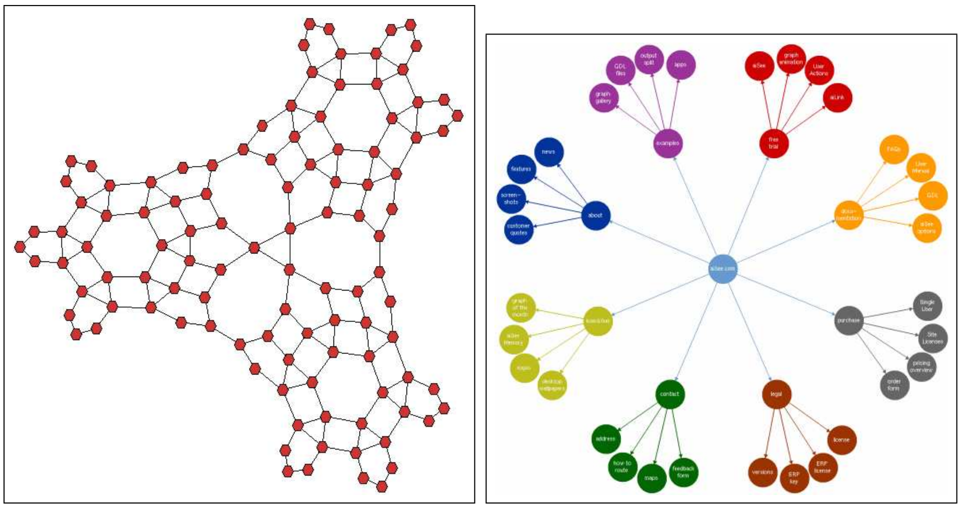

Figure 3.5: Two symmetric drawing samples. (left: courtesy of Tom Sawyer Software, right: courtesy of aiSee)

Force directed layout algorithms are very popular and successful. They reveal structural properties like symmetries, cycles and trees nicely. They have reasonably fast implementations utilizing Barnes-Hut trees and similar data structures. However force directed layout algorithms do not guarantee anything about the final drawing and unit edge length assumption may introduce serious problems for some graphs. An excellent analysis for different approaches to force directed layout is given in [11].

3.5 Previous work on compound graph layout

There has been a great deal of work done on general graph layout [11] but considerably less on layout of compound graphs [30, 5, 16], probably due to the difficult nature of the problem.

Straightforward approaches to layout compound graphs in a top-down or bottom-up (with respect to the inclusion or nesting hierarchy) manner fail, due to bidirectional dependencies (e.g., inter-graph edges) between levels of varying depth. The limited work done on compound graph layout has mostly focused on layout of hierarchical graphs [31, 30], where underlying relational information is assumed to be under a certain hierarchy (see Figures 3.6 and 3.7).

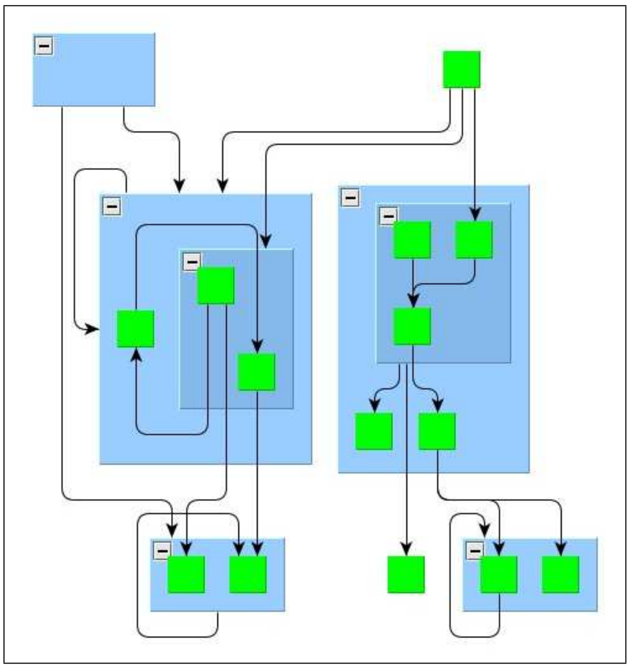

Figure 3.6: A hierarchical drawing sample with compound nodes, note that graph is directed (courtesy of yFiles).

However such algorithms perform poorly if the graph is undirected (or edge directions do not enforce a hierarchy) but still has structural properties like symmetry or include parts or substructures such as cycles. The work done on undirected compound graphs [5, 35, 19], on the other hand, is either restricted in the type of graphs addressed (e.g., nesting allowed for only one level) or unsatisfactory in terms of the quality of results produced (e.g., large compound nodes overlapping with others).

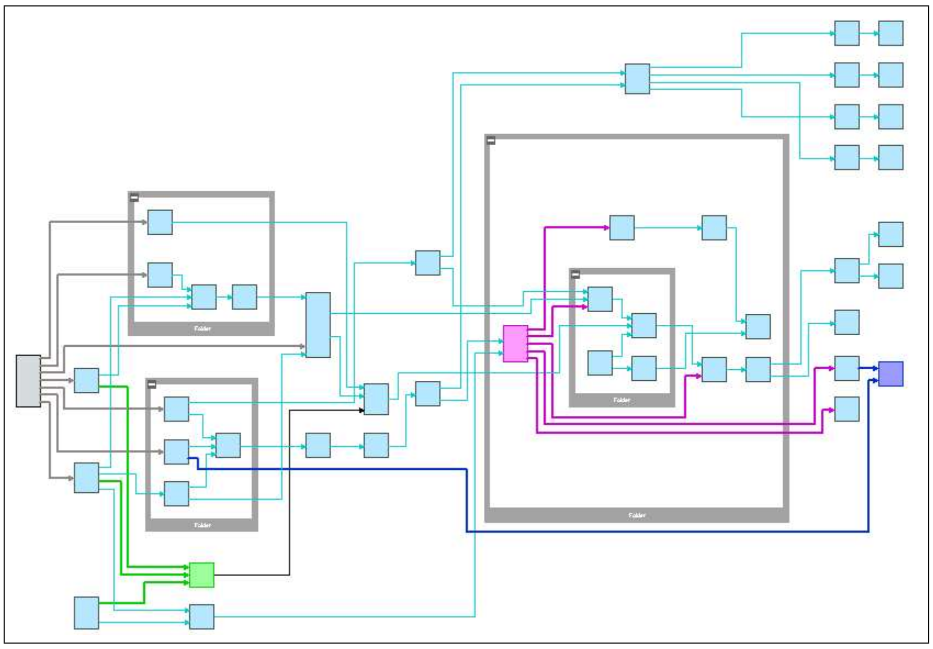

Figure 3.7: A hierarchical drawing sample with compound nodes; note that graph is directed (courtesy of Tom Sawyer Software).

Chapter 4

Layout Algorithm

4.1 Underlying Physical Model

A force-directed layout algorithm with constraints to satisfy the general drawing conventions in compound graphs has been chosen. Basic idea of the layout algorithm is to simulate a physical system in which nodes are assumed to be physical objects with certain “electrical charge”, connected via “springs” of a pre-specified desired length. Objects pull or repel each other depending on current lengths of any connected springs. In addition, relatively minor repulsion forces act on any pair of objects that are “too close” to each other to avoid node-to-node overlaps. Furthermore, each nested graph including the root of the nesting hierarchy is assumed to have a dynamic (with respect to the graph bounds) “center of gravity”. Thus the optimal layout is regarded as the state of this system, in which total energy is minimal [20]. Following additions are made to this basic model:

- An expanded node and its associated nested graph are represented as a single entity, similar to a “cart” which can move freely in orthogonal directions (no rotations allowed). Multiple levels of nesting is modeled with smaller carts on top of larger ones.
- The nodes and edges of a nested graph are to be set in motion on this

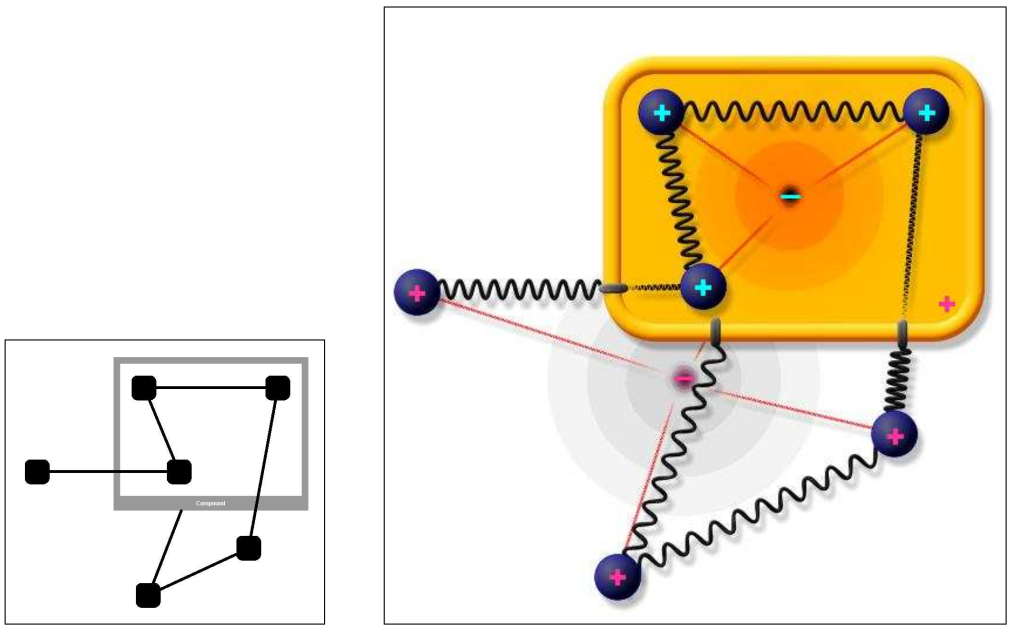

Figure 4.1: Part of a sample compound graph (left) and the corresponding physical model used by our algorithm (right).

cart, confined within the bounds of the cart. Each cart is assumed to be surrounded by a material, elastic enough to adapt to the current bounds of the associated nested graph. Thus, as nodes of a nested graph are pushed outwards, expanding the nested graph, the parent node adjusts its bounds accordingly. Similarly should the bounds of the nested graph shrink, so does the geometry of the parent node by the same amount.

- Each nested graph including the root graph is assumed to have a dynamic (with respect to its graph bounds) center of gravity, pulling all its nodes in, towards its center so as to keep them together, disallowing arbitrary drift away. Strength of this force is independent from the size of the node and the distance between node center and graph center. Similar to repulsion forces, gravitational forces are assumed to be relatively weaker than spring forces.

- In order to handle varying node sizes (especially expanded nodes) and avoid overlaps with neighboring nodes, calculation of desired edge lengths are based on the parts of edges in between borders of end-nodes, as opposed to their centers.

- Node-to-node repulsion forces take the node sizes into account. In other words, the larger a node is, the stronger it repels any node that is too close to itself. This repulsion schema can be implemented easily by calculating clipping points of the line connecting centers of nodes with the node boundaries and calculating the repulsion for these clip points instead of node centers. For simplicity and improved efficiency, two nodes repel each other only if they are within the same graph.

- Intergraph edges are treated specially; the part of an intergraph edge e, if any, from its end-node u in a nested graph G_u to the boundary of G_u is represented by a constant force (similar to gravitational forces), instead of a spring, so as to keep u as close to the boundary of G_u as possible. The remaining part of the intergraph edge is represented with a regular spring. Such a special treatment requires heavy computation and as the nesting tree gets deeper, hence the average number of graphs spanned by an intergraph edge increase, computational cost required to accurately simulate this model will raise dramatically. However it is possible to approximate this model by increasing the desired length of an intergraph edge with an amount proportional to the sum of the depths of its end nodes from their common ancestors in the nesting tree (see Figure 4.2) as inter graph edges are to cross one or more graph boundaries. If this schema is chosen, forces of intergraph edges should be propagated to ancestors of the end nodes. The latter alternative, which is an approximation to the original performs as well as the original schema in terms of quality and gives much better running time performance.

Figure 4.1 illustrates the basics of our physical model with an example.

4.2 Application-Specific Constraints

Today's sophisticated graph visualization applications require specific constraints to be integrated into layout algorithms. These constraints may vary arbitrarily,

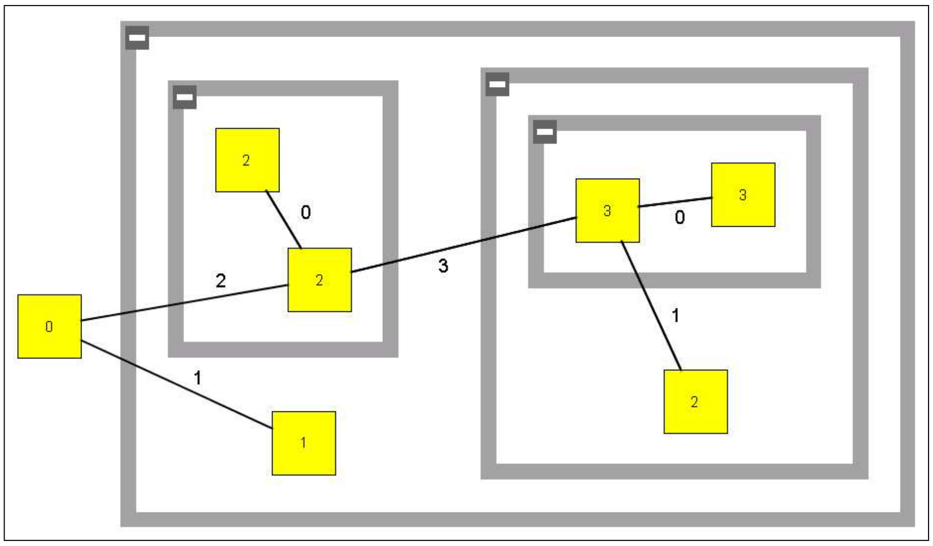

Figure 4.2: A sample graph with several inter graph edges. The ideal edge length of each edge is modified with the factor shown in edge labels. Notice that this factor is dependent on total depth from the common ancestor.

however common examples include keeping relative position of a group of nodes fixed and clustering a set of nodes that share a common property worth displaying [7]. However such constraints generally introduce conflicting goals even with the core target of the basic spring embedder itself (minimal node-node overlap and revealing symmetries).

We propose introducing additional forces to "blend" application-specific constraints into our method of drawing undirected compound graphs. In order to preserve the nice properties of the core spring embedder, in case of conflict, the default forces should govern such additional forces. Thus application-specific forces are set to have constants of relatively smaller factors.

As a case study let us consider the Patika editor. Patika [10], a pathway database and tool, is composed of a server-side, scalable, database and client-side editors to provide an integrated, multi-user environment for visualizing and manipulating network of cellular events. Patika is mainly intended for signaling pathways whose underlying graph structure can be arbitrarily more complicated and irregular than that of metabolic pathways.

For a biological pathway drawing, it is quite important to group products, substrates and effectors of a reaction. Hence we apply *relativity constraint forces* or simply *relativity forces* on each substrate, product and effector states to position them properly around their associated transition(s). The convention is to align substrates and products of a transition on opposite sides of the transition to form a certain flow direction. Effector edges, on the other hand, are left freely. When calculating relativity forces, we first determine a flow, called *orientation*, for each transition by simply looking at the current, relative positions of their associated substrates and products. Then each associated state of the transition is applied a relativity force to respect this orientation (Figure 4.3).

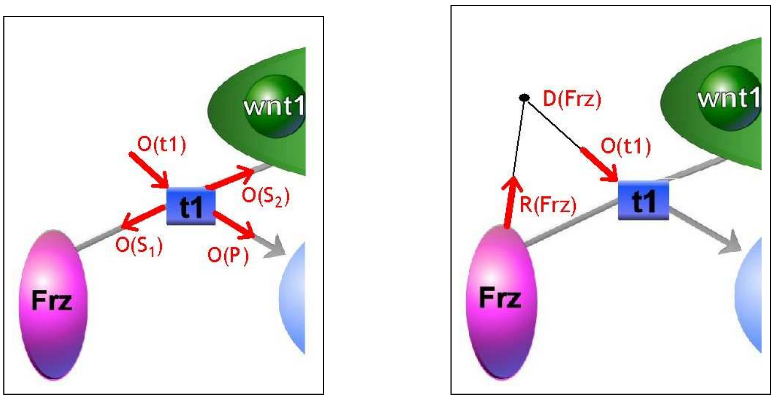

Figure 4.3: An example of how the orientation of a transition is determined shown on transition t1 of Figure 1.3 (left) and used to calculate the relativity force on one of its substrates, Frz (right). O(t1), R(Frz), and D(Frz) respectively denote orientation of t1, relativity force on Frz due to t1, and desired location of Frz to obey this force, where magnitude of R(Frz) is equal to that of O(t1), and the distance of D(Frz) from t1 is equal to the desired edge length.

Another application-specific constraint PATIKA has is due to cellular locations of biological nodes called compartments. A layout algorithm must keep each biological node within the bounds of the associated compartment and must enlarge or shrink it as required by the geometry of the enclosed part of the pathway.

The algorithm represents each compartment with a rectangular region and treats them similar to an expanded node; however unlike an expanded node, a compartment neighbors one or more other compartments and a change in its geometry affects its neighbors. So a special mechanism to resize a compartment needs to be performed. Figure 4.4 shows the layout of compartments within a cell assumed by our algorithm and used by the PATIKA editor.

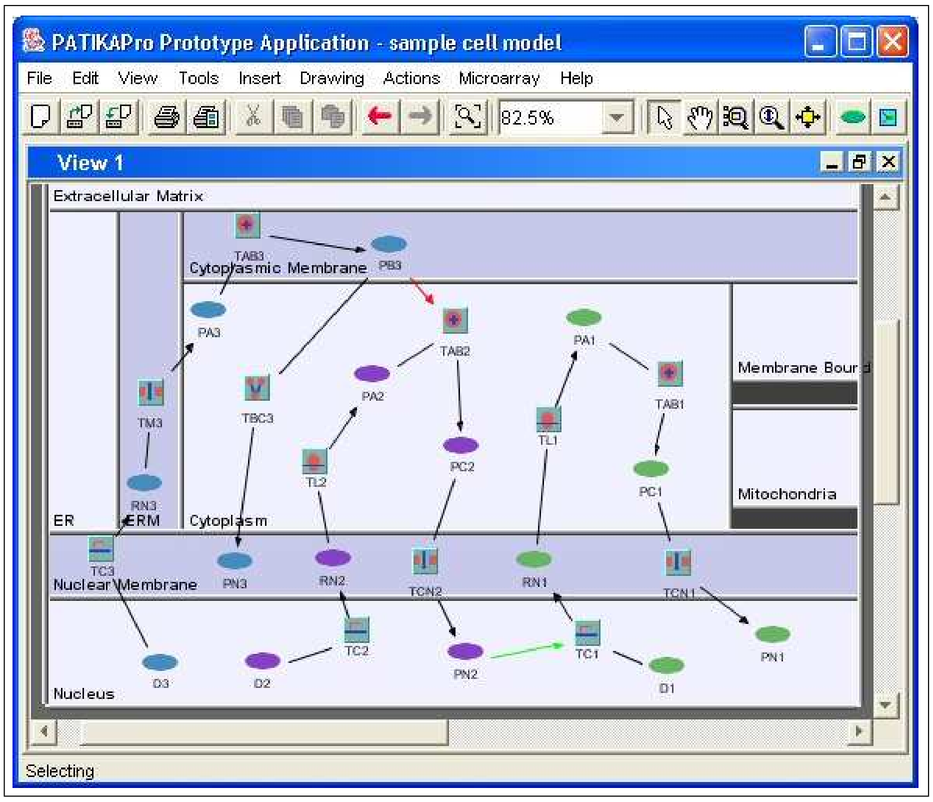

Figure 4.4: The cell model assumed by our algorithm and used by PATIKA. Each biological node is confined to its compartment.

Finally bond edges that represent the binding relations between members of a molecular complex are conventionally shorter than other interaction edges; hence we set their spring constants to be smaller.

Figure 6.9 shows a sample biological pathway drawing produced by the layout algorithm as implemented within the PATIKA editor.

4.3 Algorithm

We assume that the graph to be laid out is represented with a graph manager object $M = (S, I, F)$, where each graph $G = (V, E)$ in S is implemented using an adjacency list representation. These objects can be referenced through structures named *GraphMgr*, *Graph*, *Node*, and *Edge*. Layout specific data and functionality are assumed to be kept in these structures as well.

The algorithm is composed of three major phases preceded by an initialization phase. Please note that parameters for algorithm calls are left out to save space:

- **Initialization:** This is where initial node sizes, and threshold values for determining convergence (based on number of nodes) are calculated, and random initial positioning of nodes are performed.

  In addition, for efficiency and layout quality reasons, parts of the given graph that are trees are temporarily removed. In other words, root graph's leaf nodes are iteratively removed until no such node is left. The remaining part of the graph forms the "skeleton" of the graph (see Figure 4.5). The details of this method is given below:

```
algorithm ReduceTrees()
1) $reducedTreeRoots := \{\}$
2) for $u \in V$, where $u$ is non-reduced do
3)   if $u$ has no neighbors then
4)     mark $u$ as reduced
5)   else if $u$ has one unmarked neighbor then
6)     while $u$ is not reduced and
           ! ($u$ is compound or $u$ is member of a compound node) do
7)       mark $u$ as reduced
8)       add $u$ to $reducedTreeRoots$
9)       for $e = (u, v) \in E$ and
```

```
            $v$ is not reduced **do**
10)         remove $u$ from $reducedTreeRoots$
11)         $u := v$
12)         add $u$ to $reducedTreeRoots$
```

The overall time complexity of this method is $\Theta(|V|)$ as each node is visited $\Theta(1)$ times.

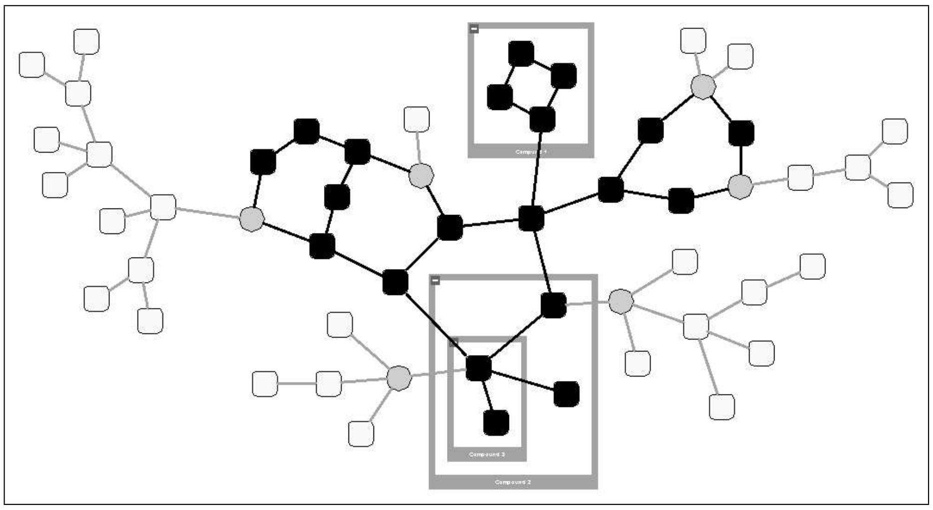

Figure 4.5: The *skeleton* is shown dark and reduced trees are marked with light color. Notice that only the trees that are members of the root graph are allowed to be reduced. Reduced tree roots are shown with circle nodes as they will be the tree growth origins for later phases of the algorithm, where trees are grown in *level-order*.

- **Phase 1:** In this phase the skeleton graph is laid out using the spring embedder model described earlier but application constraint and gravitational forces are disabled.

- **Phase 2:** Trees reduced earlier in the initialization phase are introduced back level by level in this phase, also taking application constraint and gravitational forces into account.

- **Phase 3:** This phase is the stabilization phase where we "polish" the layout.

The formula for calculating the spring force is

$$F_s = (\lambda - edgeLength)^2/\eta,$$

where λ is the ideal edge length and η is the elasticity constant of the edge. Ideal edge length of an intergraph edge is increased proportional to the sum of the depths of its end nodes from their common ancestors in the nesting tree. In addition, edges have different types based on their end-nodes being simple or compound; as compound nodes require force calculations to be based on clipping points rather than node centers. The following method is used for calculating spring forces acting on each edge's ends:

```
algorithm CalculateSpringForces(
      Graph $G = (V, E)$)
1)  for $e = (u, v) \in E$ do
2)    if e is SIMPLE-SIMPLE
3)      calculate $F_s$ for $u.center$ and $v.center$
4)    else if e is COMPOUND-SIMPLE
5)      calculate $F_s$ for $u.clipPoint(e)$ and $v.center$
6)    else if e is SIMPLE-COMPOUND
7)      calculate $F_s$ for $u.center$ and $v.clipPoint(e)$
8)    else if e is COMPOUND-COMPOUND
9)      calculate $F_s$ for $u.clipPoint(e)$ and $v.clipPoint(e)$
10)   $F_s(u)$ += $F_s$
11)   $F_s(v)$ −= $F_s$
```

The overall time complexity of this method is $\Theta(|E|)$ as all steps inside the for-loop can be processed in $\Theta(1)$ steps.

Node-to-node repulsion forces are calculated using the formula

$$F_r = \alpha/(d_x^2 + d_y^2),$$

where α is the repulsion constant and d_x and d_y are the differences in x and y coordinates of the two repulsing nodes, respectively. Similar to spring forces, repulsion forces require us to make clipping point calculations for compound nodes based on the line passing through nodes' centers:

```
 algorithm ApplyRepulsionForces(
       Graph $G = (V, E)$)
1)  $S := \{\}$
2)  for $u \in V$ do
3)    add $u$ into $S$
4)    if $u$ is a compound node then
5)      $c_u :=$ clipping point of the $line(u.center, v.center)$ and $u.boundRect$
6)    else
7)      $c_u := u.center$
8)    for $v \in V - S$ do
9)      if $v$ is a compound node then
10)       $c_v :=$ clipping point of the $line(u.center, v.center)$ and $v.boundRect$
11)     else
12)       $c_v := v.center$
13)     if $dist(c_u, c_v) < repulsionRange$ then
14)       Calculate repulsion force $F_r$ for $c_u$ and $c_v$
15)       $F_r(u)$ += $F_r$
16)       $F_r(v)$ −= $F_r$
```

Steps 9-16 are handled in $\Theta(1)$ steps, which are executed a total of maximum $O(|V|^2)$ times, making the overall complexity of the method $O(|V|^2)$. However, since a node pair affect each other only when they are below a certain geometric distance, the average complexity is expected to be asymptotically lower than this.

Gravitation forces have fixed magnitude and they are always towards the center of the bounding rectangle of the owner graph:

```
 algorithm ApplyGravitationForces(
       Graph $G = (V, E)$)
1) for $u \in V$ do
2)    $center := u.ownerGraph.boundRect$
3)    calculate gravitation force $F_g$ towards $center$
4)    $F_g(u) \mathrel{+}= F_g$
```

The overall time complexity of this method is $\Theta(|V|)$ as all steps inside the for-loop can be processed in $\Theta(1)$ time.

The following method is used for calculating the relativity constraint forces introduced in our case study as an example of application-specific force calculation:

```
 algorithm ApplyAppSpecificForces(
       Edge $e = (u, v)$)
1) if $phase \geq 2$ then
2)    $orientation := e.transition.orientation$
3)    if e is substrate then
4)       $orientation := -orientation$
5)    Calculate $F_{rc}$ on $e$ according to its orientation
6)    $F_s(u) \mathrel{+}= F_{rc}$
7)    $F_s(v) \mathrel{-}= F_{rc}$
```

The main method making use of earlier ones to implement the layout algorithm is as follows:

```
 algorithm CompoundLayout()
1) call Initialize()
2) $phase := 1$
3) if layout type is incremental then
4)    $phase := 3$
```

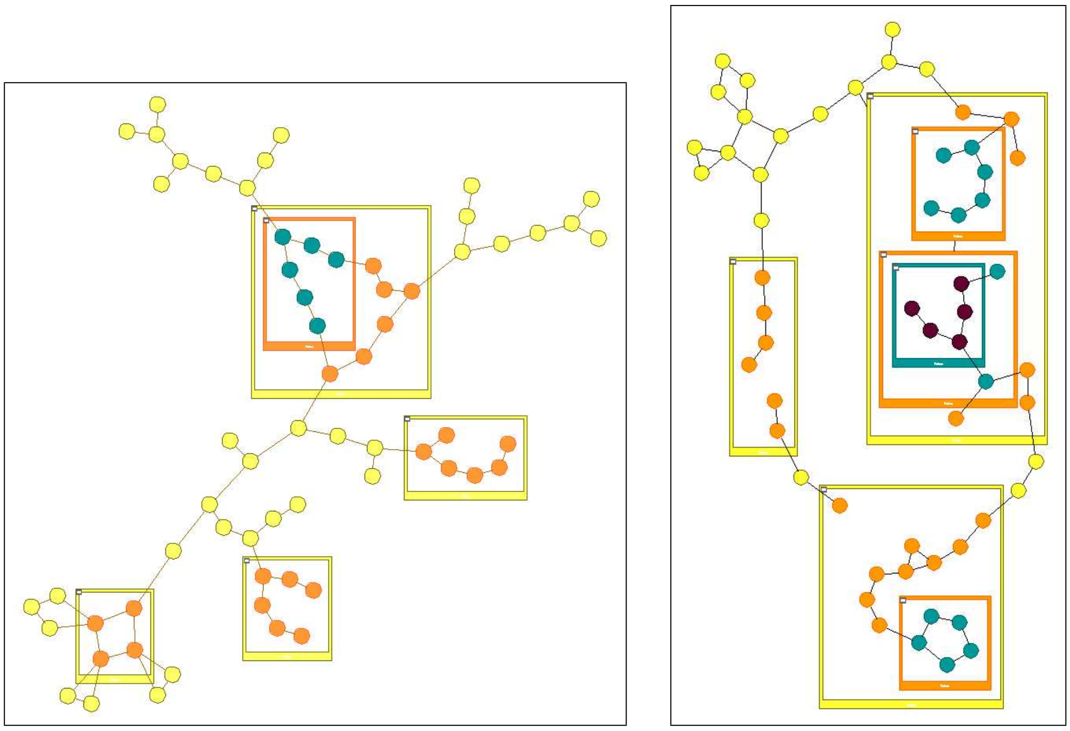

Figure 4.6: Sample compound graphs (with varying desired edge lengths and edge and intergraph edge density) laid out by our algorithm. The nodes are color-coded to denote the depth of the node in the nesting hierarchy (i.e., the deeper a node is, the darker its color is).

```
5)  **while** $phase \leq 3$ **do**
6)    $step := 1$
7)    $error := 0$
8)    **while** ($step < maxIterCount(phase)$ **and**
          $error > errorThreshold(phase)$) **or** $!allTreesGrown$ **do**
9)      **call** APPLYSPRINGFORCES()
10)     **call** APPLYREPULSIONFORCES()
11)     **if** $phase \neq 1$ **then**
12)       **call** APPLYGRAVITATIONFORCES()
13)       **call** APPLYAPPSPECIFICFORCES()
14)     **call** CALCNODEPOSITIONSANDSIZES()
15)     **if** $phase = 2$ **and** $!allTreesGrown$ **and**
            $step \% treeGrowingStep = 0$ **then**
16)       **call** GROWTREESONELEVEL()
```

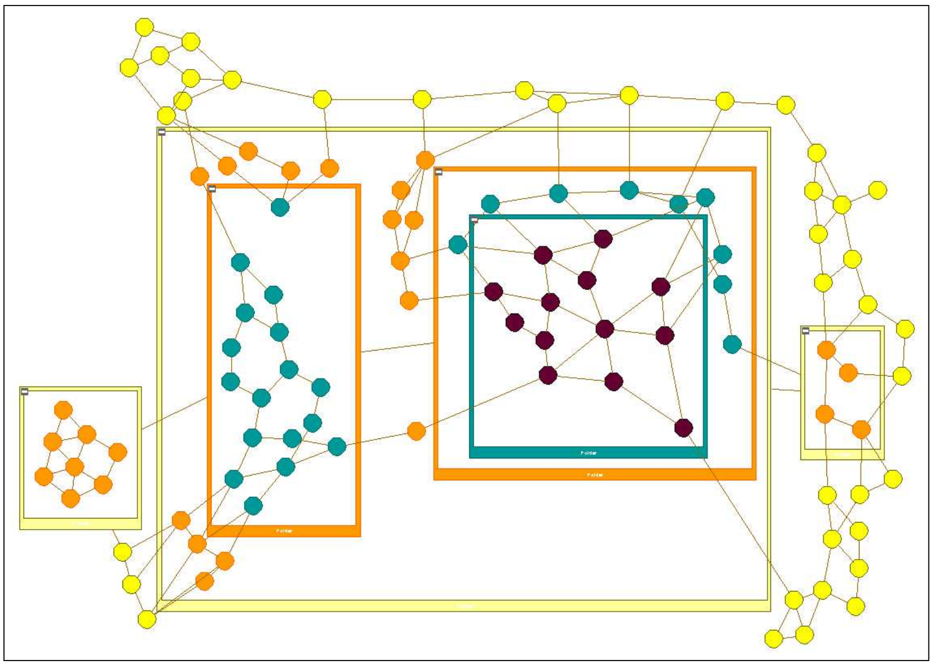

Figure 4.7: Another sample compound graphs laid out by our algorithm. Notice there are dense mesh like structures with many intergraph edges.

```
17)     $step := step + 1$
18)  $phase := phase + 1$
```

A quick analysis of the algorithm reveals that the running time of layout of a compound graph is $O(k \cdot n^2)$ where n is the total number of nodes in the compound graph, and k is the number of iterations required to reach an energy minimal state.

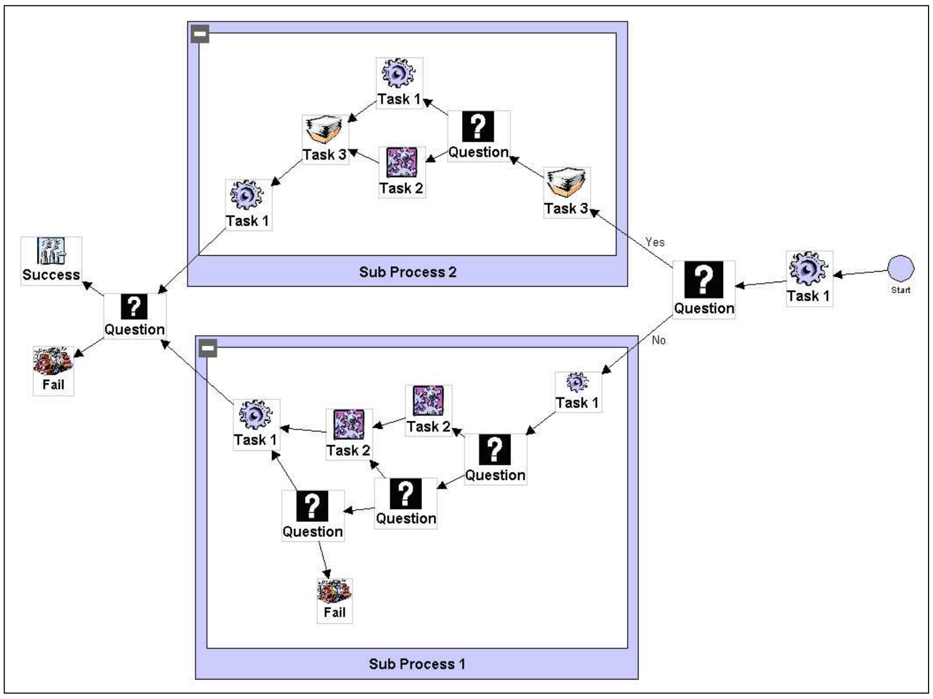

Figure 4.8: A sample business work flow graph (courtesy of Tom Sawyer Software) laid out by our algorithm.

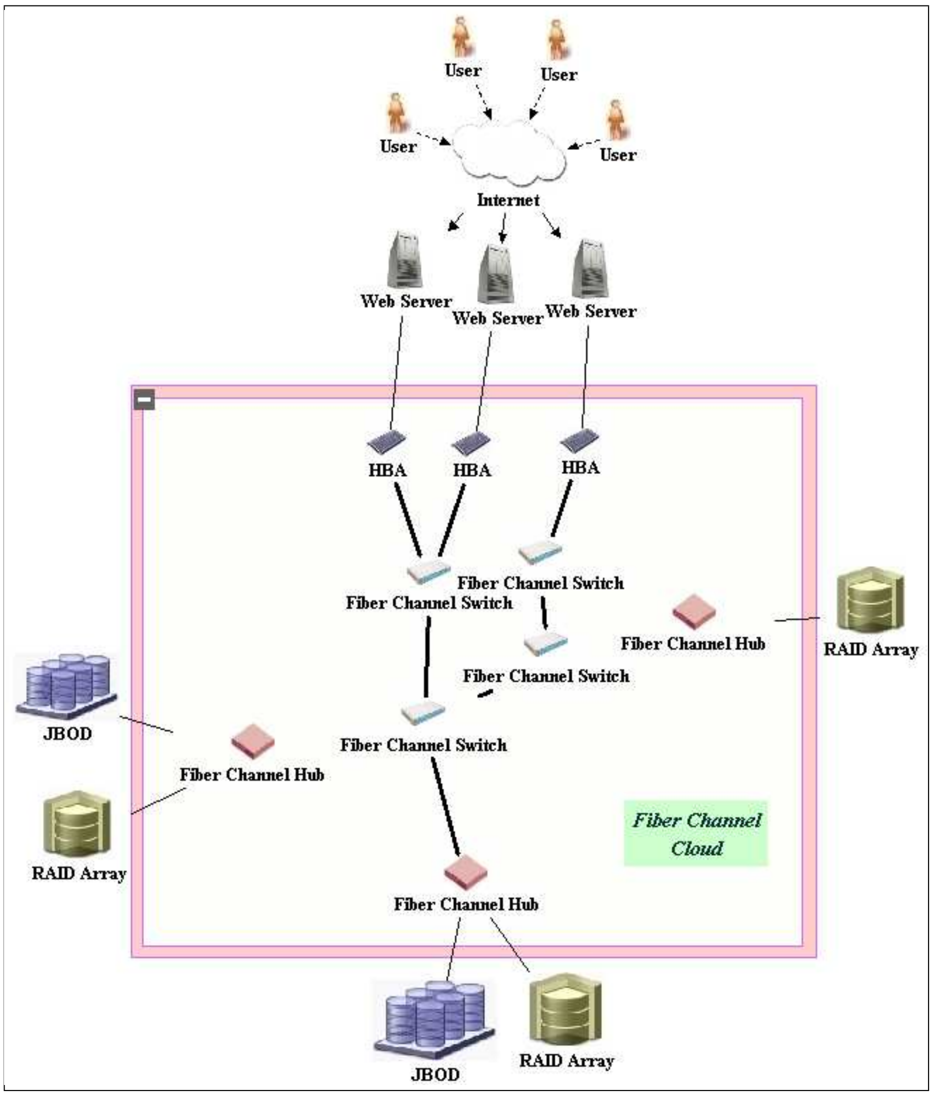

Figure 4.9: A sample networking graph (courtesy of Tom Sawyer Software) laid out by our algorithm.

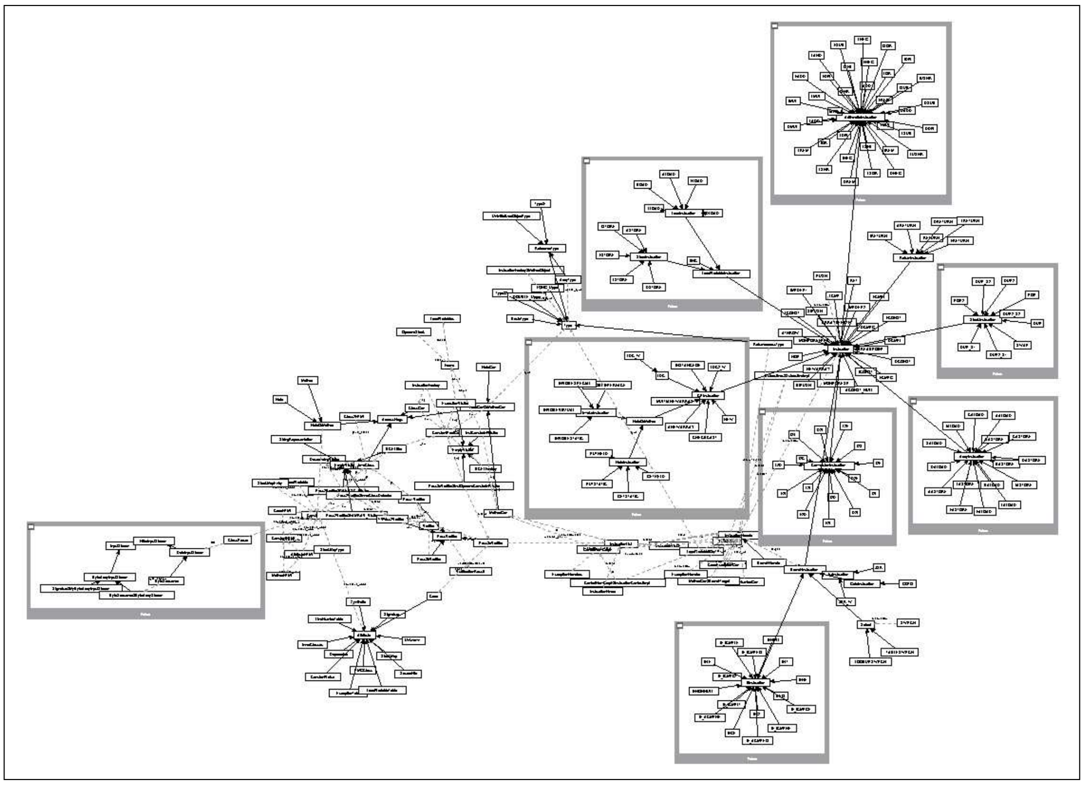

Figure 4.10: A sample software modeling graph (courtesy of Tom Sawyer Software) laid out by our algorithm.

Chapter 5

Implementation

The proposed layout algorithm has been tested within the example application of Tom Sawyer Visualization for Java, version 7.0. The development environment was Sun's Java SDK 1.4 and Microsoft Windows XP operating system on an ordinary personal computer (Pentium IV with 2GHz CPU and 512MB memory). The results have been found quite satisfactory as far as the general graph drawing criteria such as number of crossings and total area are concerned (Figures 4.6 and 4.9). Furthermore, the experimental executions have been found not only reasonably fast for interactive use but also in line with the earlier theoretical analysis as detailed out below.

5.1 An Application

We have also implemented our algorithm as part of a new version of the PATIKA pathway editor.

It is expensive to move real PATIKA objects throughout the layout procedure. Moreover during certain sub-phases of layout we only want to deal with a subset of the view graph. Therefore a light-weight abstraction that captures the geometry of subject view is necessary (see Figure 5.1).

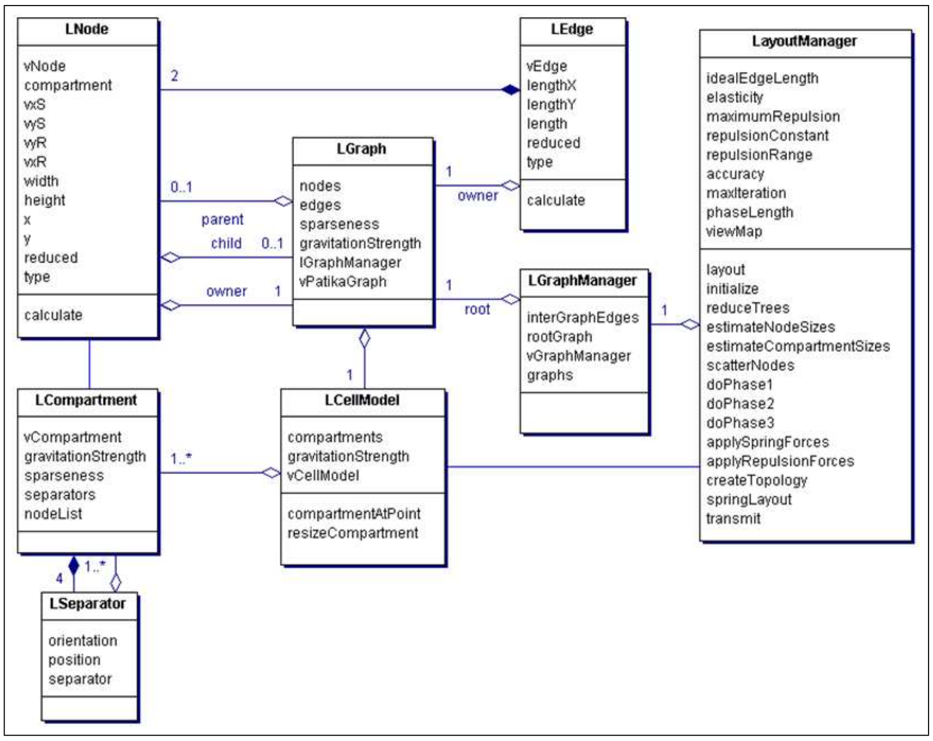

Figure 5.1: All information required to layout a view is stored as described here.

To be able construct such abstraction for layout manager to operate on; the subject view of layout operations is analyzed and all geometric information is collected. This information is used to fill in the class hierarchy shown above. Layout manager exclusively runs on this abstraction and the resulting geometric changes are propagated to the view periodically if not at once after layout is finished.

The results have been found satisfactory as far as the general graph drawing criteria such as number of crossings and total area are concerned. In addition, application-specific constraints such as compartmental constraints and relative positioning constraints seem to be highly satisfied. Figure 6.9 shows a sample pathway drawing produced. Notice that subcellular location (i.e., compartments) of biological nodes are respected as well as grouping (i.e., nesting), and compartments are resized to tightly fit their contents.

5.2 Implementation Details

During the implementation we were faced with certain issues, around which we had to make adjustments to improve our algorithm.

Special intergraph edge treatment as described earlier is difficult to implement. Instead we have chosen another schema that seems to work just as well. Let $e = (x, y)$ be an intergraph edge and l be the sum of the depth of the owner graphs of x and y from their common ancestors in the nesting tree. The desired edge length of e is set to be longer than the desired edge length of a regular edge by an amount linearly proportional to l since such edges need to additionally cross the associated nested graph boundaries.

In addition, to emulate the constant boundary forces on intergraph edges discussed earlier, spring forces calculated for intergraph edges are reflected to the ancestor nested graphs in the nesting tree, starting from owners of the intergraph edge's end-nodes, up until their common ancestor. The magnitude of this force, however, decreases as we go up the nesting tree.

Furthermore, we have limited the movement of each node in each iteration to avoid drastic movements, often resulting in "oscillations".

Finally, the use of "momentum" or "temperature" for each node [18] has helped the convergence greatly. Each node's movement is not only based on the total force calculated during the current iteration but also on the previous one. For simplicity and efficiency reasons we have simply added a constant portion of the previous iteration's total force to this iteration's total force for each node, resulting in dramatic improvements in execution times.

Chapter 6

Experimental Results

6.1 Running time performance

We have performed experiments on execution time of our layout algorithm on randomly generated graphs with one of several parameters changing for each set. For each test, a random graph manager to be laid out has been generated with the provided parameters:

- n: total number of nodes,
- m/n: proportion of number of edges to nodes; the number of edges is assumed to be linear in the number of nodes,
- m_{ig}/m: proportion of intergraph edges to number of all edges,
- d: maximum nesting depth,
- b: maximum branching (i.e., number of children of a node) in the nesting tree,
- p: probability of pruning a child in the nesting tree to avoid nesting trees that are too uniform in structure.

First we construct a nesting tree and a graph manager that realizes this nesting structure with the specified parameters. Then the nodes are created and distributed to graphs in the graph manager uniformly. Similarly end nodes of each edge is picked randomly. Each test is executed 10 times and the average is taken. Figure 6.1 shows an example of a randomly generated graph.

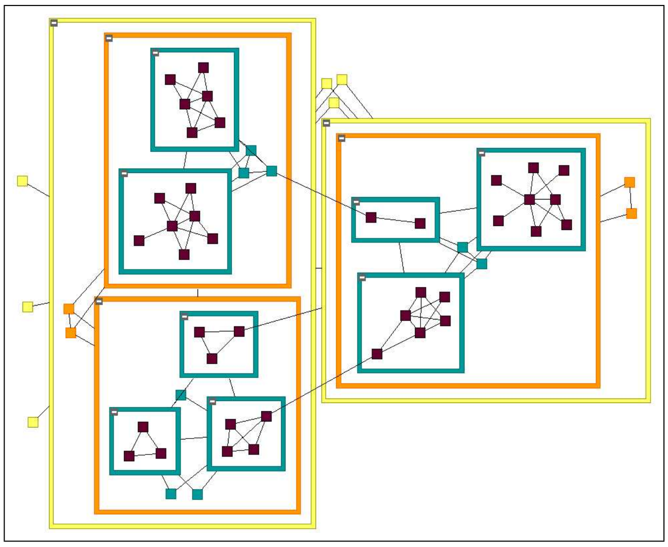

Figure 6.1: A randomly generated graph laid out by our algorithm. ($n = 70$, $m/n = 1.5$, $m_{ig}/m = 0.03$, $d = 3$, $b = 3$, and $p = 0.33$)

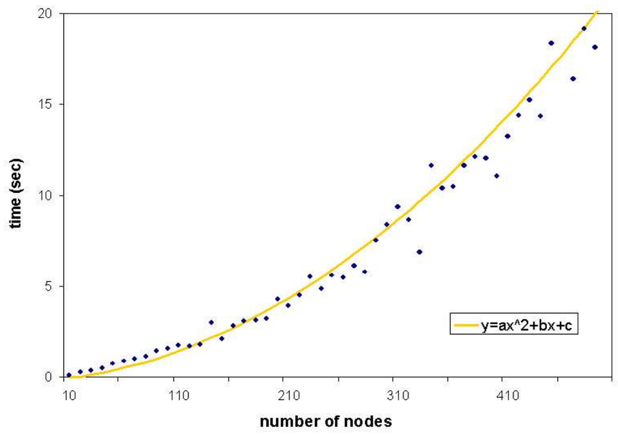

Figure 6.2: Number of nodes (n) vs. execution time of our algorithm. ($m/n = 1.5$, $m_{ig}/m = 0.05$, $d = 3$, $b = 3$, and $p = 0.33$)

From the theoretical analysis given earlier, a quadratic behavior of execution time is expected, assuming k does not grow in the order of the graph size. The experiments validate this argument (Figures 6.2).

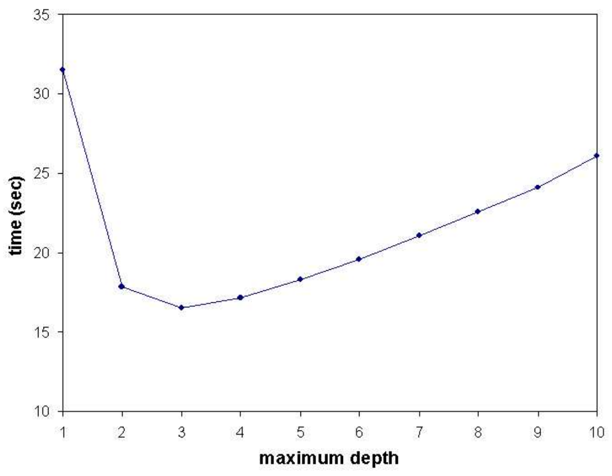

Figure 6.3: Maximum nesting depth (d) vs. execution time of our algorithm. ($n = 500$, $m/n = 1.5$, $m_{ig}/m = 0.05$, $b = 3$, and $p = 0.33$)

We have also experimented with the nesting depth (Figures 6.3). The experiments show that initially deeper nesting helps improve execution time as number of nodes per graph decreases due to the fact that certain calculations such as node-to-node repulsion forces are only performed within each graph. However as the nesting depth increases the performance decreases dramatically due to the increase in the number of compound nodes and nested graphs.

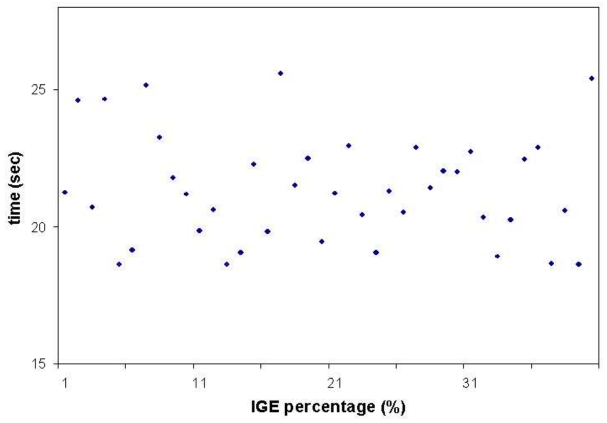

Figure 6.4: Proportion of intergraph edges to all edges (m_{ig}/m) vs. execution time of our algorithm. ($n = 500$, $m/n = 1.5$, $d = 3$, $b = 3$, and $p = 0.33$)

Furthermore, we have performed a test set to see how the proportion of intergraph edges to regular edges affect the execution time (Figures 6.4). As expected the time it takes to process an intergraph edge as opposed to a regular edge does not vary much.

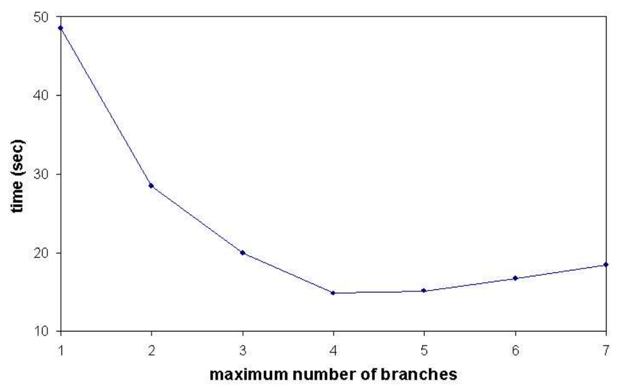

Figure 6.5: Maximum branching in the nesting tree (b) vs. execution time of our algorithm. ($n = 500$, $m/n = 1.5$, $m_{ig}/m = 0.05$, $d = 3$, and $p = 0.33$)

Lastly we wanted to see how the average number of nested graphs per graph affected the execution time (Figures 6.5). Again, initially deeper nesting helps decrease the execution time since some expensive calculations are then performed in a divide-and-conquer fashion. However, as the nesting becomes even deeper, the time it takes to process more compound nodes and deeper nodes dominate and the execution gets slower.

6.2 Quality

In our experiments, quality of the resulting layout is also inspected. According to commonly accepted layout quality criteria, results are found quite satisfactory. In most of the cases edge lengths are uniform, space is used wisely and node-node overlap is prevented (Figures 6.6, 6.7, 6.8, 6.9).

Also PATIKA specific constraints are handled quite nicely, note that compartment sizes are just adequate to hold compartment members, while compartment neighborhood information is preserved (Figures 6.7, 6.9).

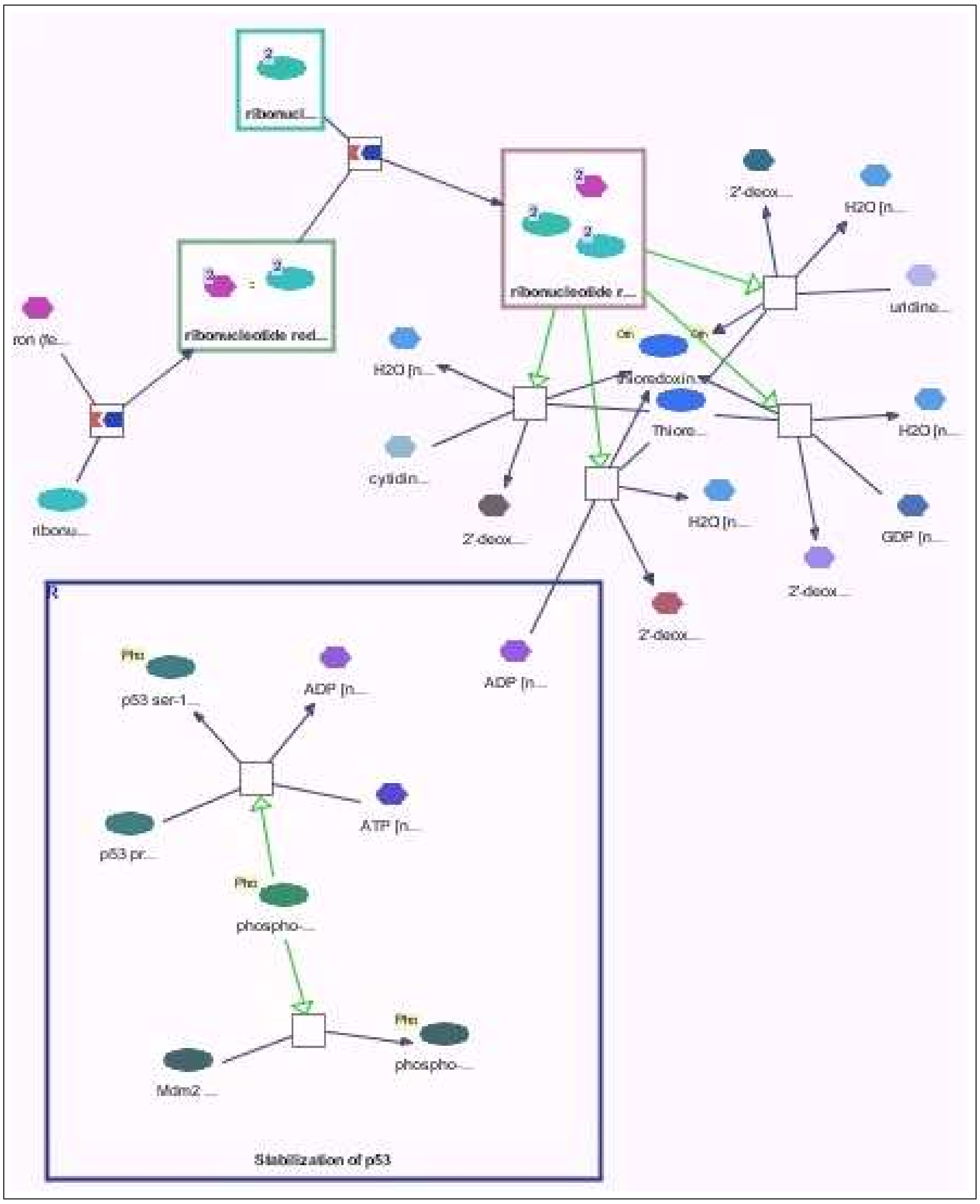

Figure 6.6: A sample pathway (stabilization of p53 pathway as obtained from the PATIKA database), laid out by our algorithm.

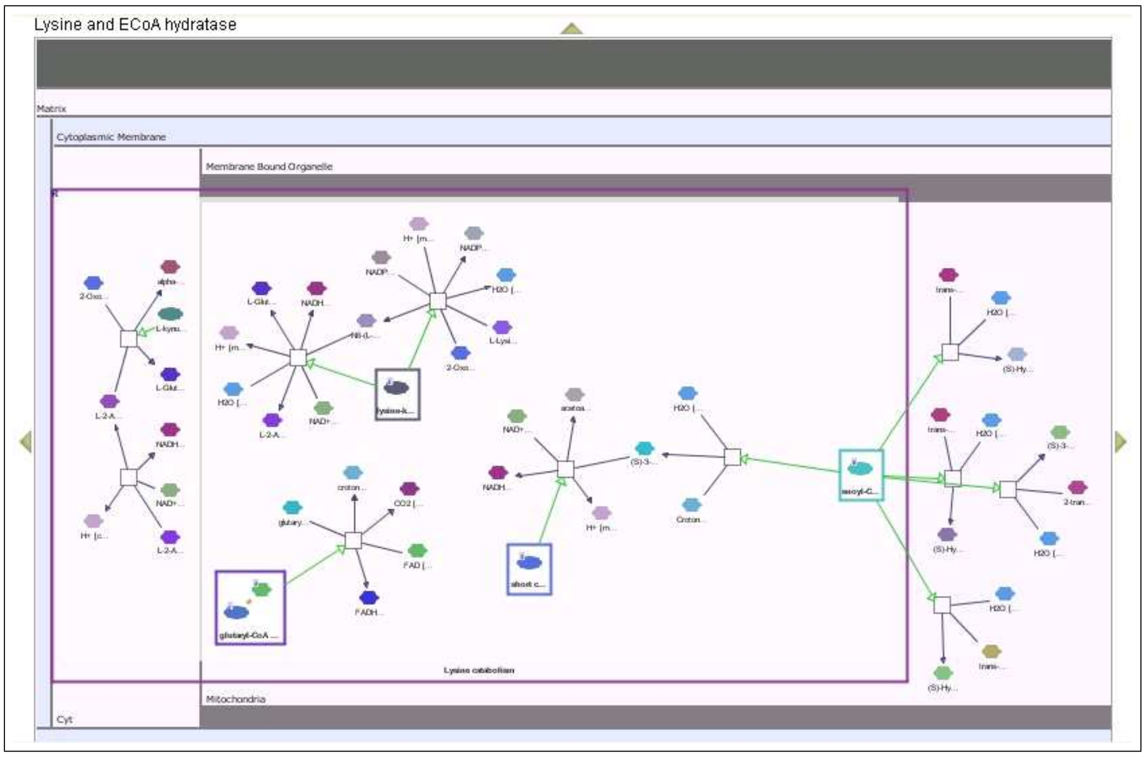

Figure 6.7: A sample pathway (Lysine and ECoA hydratase pathway as obtained from the PATIKA database), laid out by our algorithm.

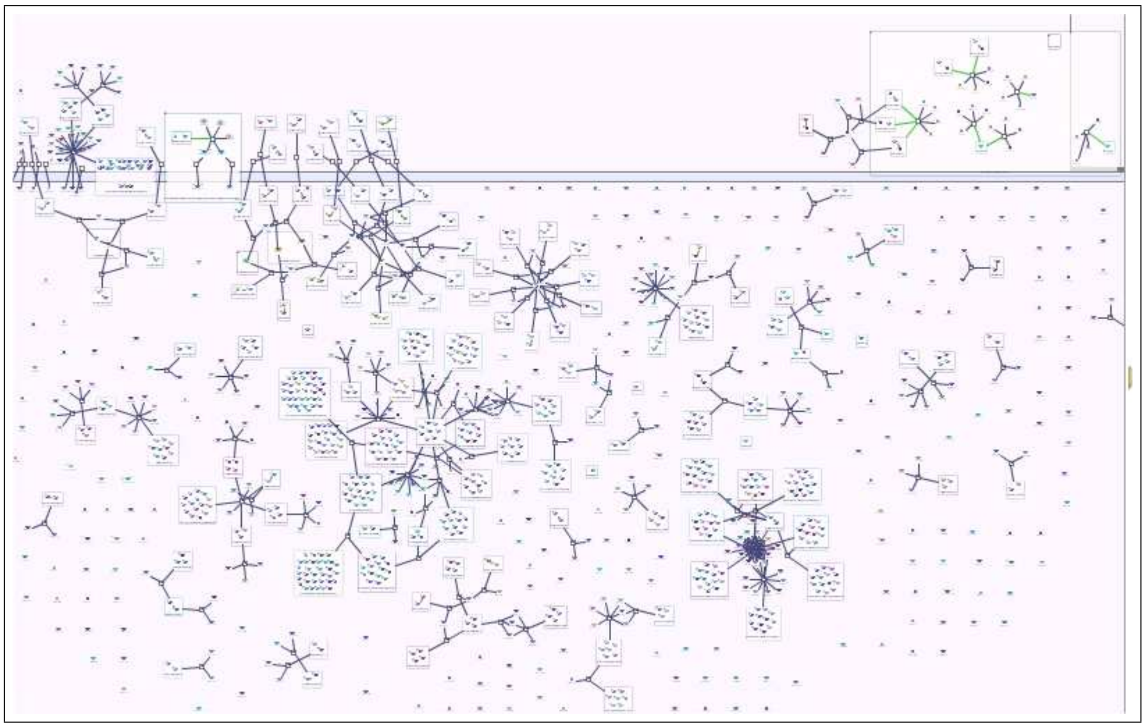

Figure 6.8: Part of a very big pathway as obtained from the Patika database, laid out by our algorithm.

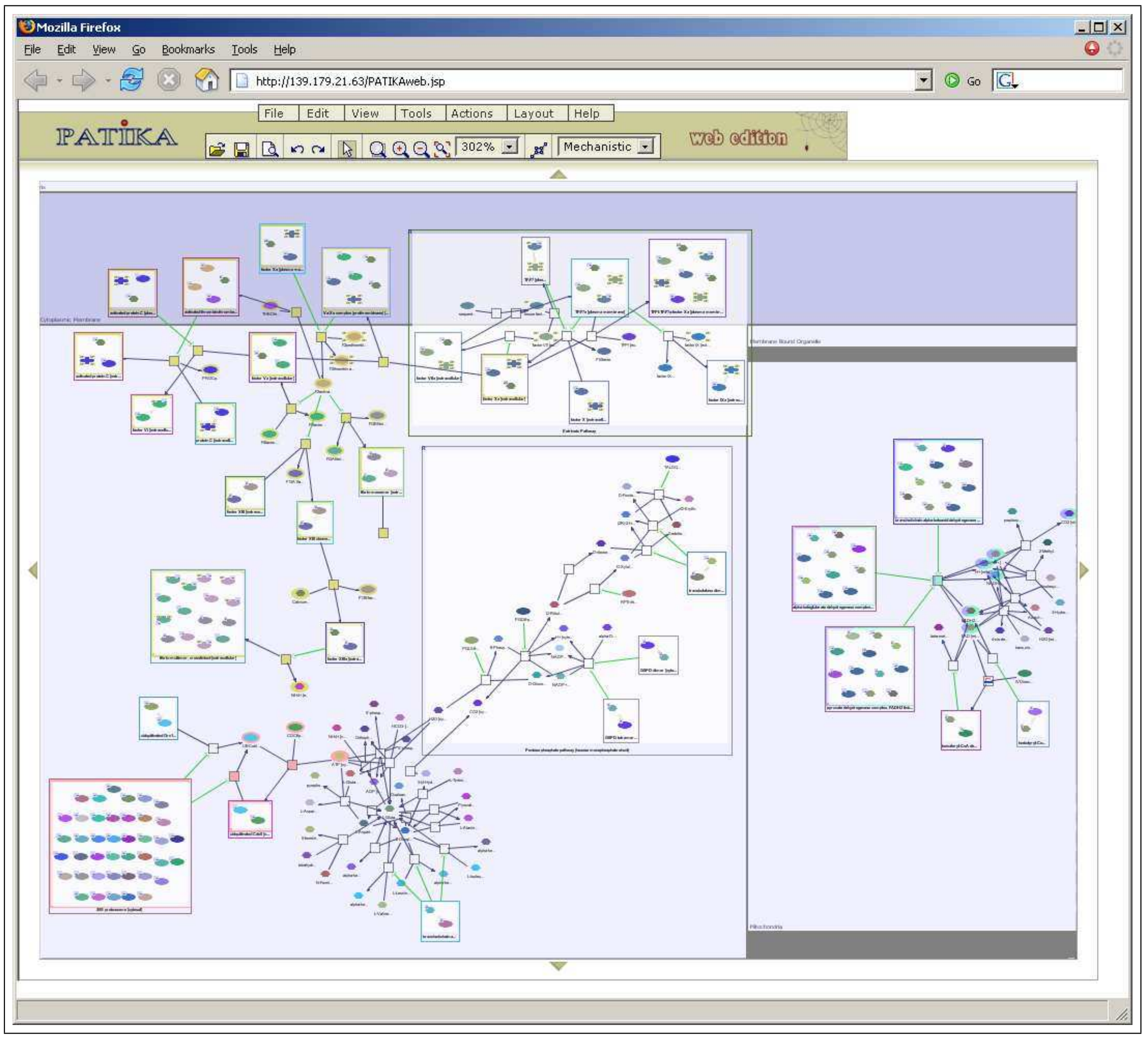
Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://139.179.21.63/PATIKAweb.jsp
Go
PATIKA
File Edit View Tools Actions Layout Help
302%
Mechanistic
web edition

Figure 6.9: A sample pathway (as obtained by several consecutive queries from the Patika database) laid out by our algorithm.

Chapter 7

Conclusion

We have presented a novel algorithm for layout of undirected compound graphs. To our knowledge, this is the first spring embedder that can handle compound graphs. Layout of complicated pathway graphs such as those in PATIKA are among the target applications. The main novelty of our work is the use of a modified spring embedder system that treats compound nodes and intergraph edges as part of a physical system. In addition, our model is quite flexible; most application-specific drawing conventions such as those in biological pathways can be easily integrated into this physical system as additional constraints. Experimental results have been found satisfactory both in terms of quality of layouts and computational efficiency.

However there is still place for improvement. As future work, we may look at ways to speed up our algorithm, as it is destined to be part of interactive systems. We should investigate efficient ways to calculate repulsion forces as it is the bottleneck. There are very popular data structures in the literature to solve n-body problems efficiently, like Barnes-Hut trees and the K-D trees [4]. However we cannot define our nodes as simple points in the space as compound nodes have variable size and aspect ratio, hence we could not make use of such advanced data structures as is. Another improvement would be to come up with a more advanced convergence criterion.

Bibliography

[1] Drawing graphs with Graphviz. http://www.graphviz.org.

[2] Govisual Software Libraries. oreas GmbH., Köln, Germany. http://www.oreas.com/.

[3] aisee User Manual. AbsInt Angewandte Informatik GmbH., Saarbruecken, Germany, 2000-2005. http://www.aisee.com/.

[4] R. J. Anderson. Tree data structures for N-body simulation. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1996.

[5] F. Bertault and M. Miller. An algorithm for drawing compound graphs. In J. Kratochvil, editor, *Graph Drawing (Proc. GD '99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 197–204. Springer-Verlag, 1999.

[6] T. Biedl, B. Madden, and I. Tollis. The three-phase method: A unified approach to orthogonal graph drawing. In G. DiBattista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 391–402. Springer-Verlag, 1998.

[7] K. Bohringer and F. Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *CHI '90 Proceedings*, pages 43–51. ACM, 1990.

[8] S. Bridgeman, J. Fanto, A. Garg, R. Tamassia, and L. Vismara. Interactivegeiotto: An algorithm for interactive orthogonal graph drawing. In G. DiBattista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 303–308. Springer-Verlag, 1998.

[9] T. Chan, M. Goodrich, S. Kosaraju, and R. Tamassia. Optimizing area and aspect ratio in straight-line orthogonal tree drawings. In S. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 63–75. Springer-Verlag, 1997.

[10] E. Demir, O. Babur, U. Dogrusoz, A. Gursoy, G. Nisanci, R. Cetin-Atalay, and M. Ozturk. PATIKA: An integrated visual environment for collaborative construction and analysis of cellular pathways. *Bioinformatics*, 18(7):996–1003, 2002.

[11] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing, Algorithms for the Visualization of Graphs.* Prentice-Hall, 1999.

[12] U. Dogrusoz, Q. Feng, B. Madden, M. Doorley, and A. Frick. Graph visualization toolkits. *IEEE Computer Graphics and Applications*, 22(1):30–37, January/February 2002.

[13] U. Dogrusoz and B. Genc. A multi-graph approach to complexity management in interactive graph visualization. *Accepted for publication: Computers & Graphics.*

[14] U. Dogrusoz, B. Madden, and P. Madden. Circular layout in the Graph Layout Toolkit. In S. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 92–100. Springer-Verlag, 1997.

[15] C. Duncan, M. Goodrich, and S. Kobourov. Balanced aspect ratio trees and their use for drawing very large graphs. In S. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 111–124. Springer-Verlag, 1998.

[16] P. Eades, Q. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In S. North, editor, *GD '96*, volume 1190 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag, 1997.

[17] P. Eades and Q. W. Feng. Multilevel visualization of clustered graphs. In S. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 101–112. Springer-Verlag, 1997.

[18] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In R. Tamassia and I. Tollis, editors, *GD '94*, volume 894 of *Lecture Notes in Computer Science*, pages 388–403. Springer-Verlag, 1995.

[19] Y. Frishman and A. Tal. Dynamic drawing of clustered graphs. In *Proceedings of IEEE Symposium on Information Visualization*, pages 191–198, October 2004.

[20] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software Practice and Experience*, 21(11):1129–1164, 1991.

[21] K. Fukuda and T. Takagi. Knowledge representation of signal transduction pathways. *Bioinformatics*, 17(9):829–837, 2001.

[22] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[23] JViews User's Guide. 94253 Gentilly Cedex, France, 2002. http://www.ilog.com.

[24] K. Sugiyama and K. Misue. A Generic Compound Graph Visualizer/Manipulator: D-ABDUCTOR. In F. J. Brandenburg, editor, *GD '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 500–503. Springer-Verlag, 1995.

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

[26] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer, 2001.

[27] W. Lai and P. Eades. A graph model which supports flexible layout functions. Technical Report 96-15, Callaghan 2308, Australia, 1996.

[28] M. Raitner. HGV: A library for hierarchies, graphs, and views. In M. Goodrich and S. Kobourov, editors, *Proc. Graph Drawing '02*, volume 1528 of *LNCS*, pages 236–243, 2002.

[29] G. Sander. Graph Layout Through the VCG Tool. In R. Tamassia and I. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pages 194–295. Springer-Verlag, 1995.

[30] G. Sander. Layout of compound directed graphs. Technical Report A/03/96, University of Saarlandes, CS Dept., Saarbrücken, Germany, 1996.

[31] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man and Cybernetics*, 21(4):876–892, 1991.

[32] K. Sugiyama and K. Misue. A simle and unified method for drawing graphs: Magnetic-spring algorithm. In R. Tamassia and I. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pages 364–375. Springer-Verlag, 1995.

[33] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(2):109–125, February 1981.

[34] Graph Layout Toolkit and Graph Editor Toolkit User's Guide and Reference Manual. Tom Sawyer Software, Oakland, CA, USA, 1992-2005. http://www.tomsawyer.com.

[35] X. Wang and I. Miyamoto. Generating customized layouts. In F. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 504–515. Springer-Verlag, 1995.

[36] yFiles User's Guide. yWorks GmbH, D-72076 Tübingen, Germany, 2002. http://www.yworks.de.