A Layout Algorithm For Undirected Compound Graphs

U. DOGRUSOZ, E. GIRAL, A. CETINTAS, A. CIVRIL and E. DEMIR
Bilkent Center for Bioinformatics

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.

Categories and Subject Descriptors: I.3.8 [**Computer Graphics**]: Applications; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; H.4.3 [**Information Systems Applications**]: Communications Applications

General Terms: Algorithms

Additional Key Words and Phrases: Information Visualization, Graph Drawing, Force-Directed Graph Layout, Compound Graphs

1. INTRODUCTION

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 [Dogrusoz et al. 2002]. Effective analysis of the underlying data in graph visualization is only possible with sound automatic layout capabilities of such systems.

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 [K. Sugiyama and K. Misue 1995; Raitner 2002; Lai and Eades 1996; Fukuda and Takagi 2001] (see Figures 1 and 2).

There has been a great deal of work done on general graph layout [Di Battista et al. 1999] but considerably less on layout of compound graphs, 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 [Sugiyama and Misue 1991; Sander 1996; Eades et al. 1997], where underlying relational information is assumed to be under a certain hierarchy. 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 [Bertault and Miller 1999; Wang and Miyamoto 1995; Frishman and Tal 2004], on the other hand, is either restricted in the type

U. Dogrusoz is with Bilkent University (Computer Eng. Dept., Ankara 06800, Turkey). An earlier, short version of this paper appeared in the Proceedings of the 12th International Symposium on Graph Drawing, NYC, NY, pp. 442-447, Sept. 29 Oct. 2, 2004.

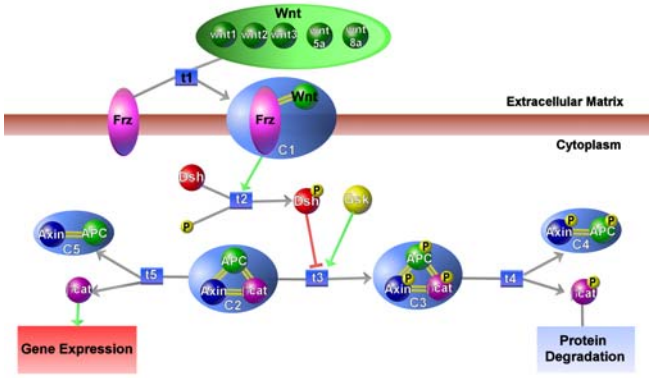

Fig. 1. Part of a sample compound pathway.

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 overlaping with others).

In this paper we describe a new algorithm for layout of undirected compound graphs, which overcomes shortcomings of previous ones. Our algorithm is based on the force-directed (spring embedder) layout algorithm [Eades 1984; Fruchterman and Reingold 1991] with arbitrary nesting relations, varying node sizes and any additional forces to handle application-specific constraints. This algorithm has also been implemented within a software tool named PATIKA [Demir et al. 2002] to visualize complicated biological pathways with compartmental constraints and nested drawings.

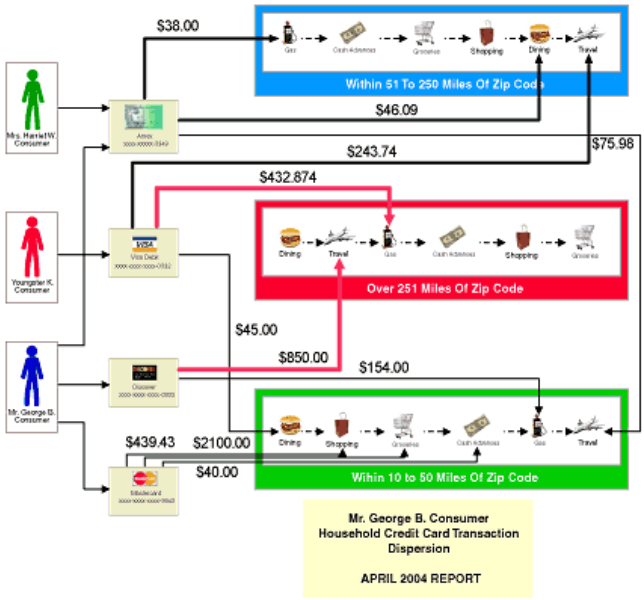

Fig. 2. A financial chart (courtesy of Tom Sawyer Software).

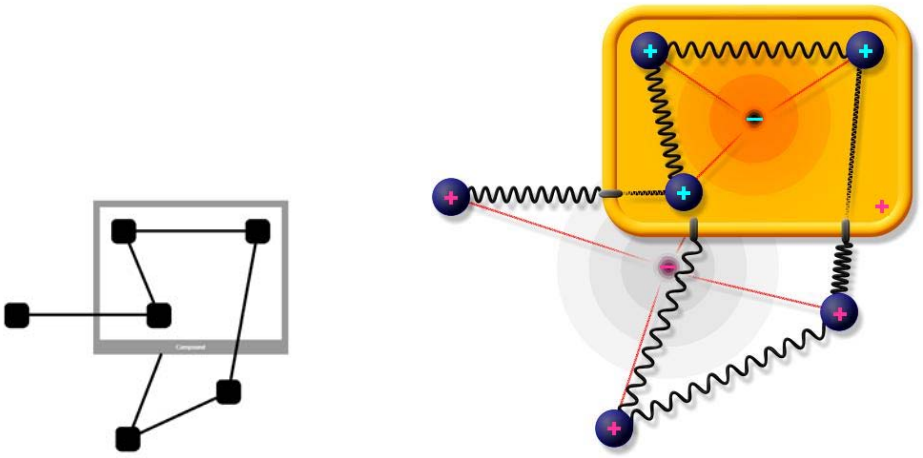

Fig. 3. Part of a sample compound graph (left) and the corresponding physical model used by our algorithm (right).

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* $F = (V^F, E^F)$. 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 $G_i \neq G_j$ 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.

3. LAYOUT ALGORITHM

3.1 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 [Fruchterman and Reingold 1991]. 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 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. 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. As the nesting tree gets deeper, the average number of graphs spanned by an intergraph edge increases, computational cost required to accurately implement 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. The latter strategy has shown as effective as the original schema in terms of quality and has yielded much better running time performance.

Figure 3 illustrates the basics of our physical model with an example.

3.2 Application-Specific Constraints

Today's sophisticated graph visualization applications require specific constraints to be integrated into layout algorithms. These constraints may vary arbitrarily, 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 [Bohringer and Paulisch 1990]. 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 [Demir et al. 2002], 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).

Fig. 4. An example of how the orientation of a transition is determined shown on transition t1 of Figure 1 (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 5 shows the layout of compartments within a cell assumed by our algorithm and used by the PATIKA editor.

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

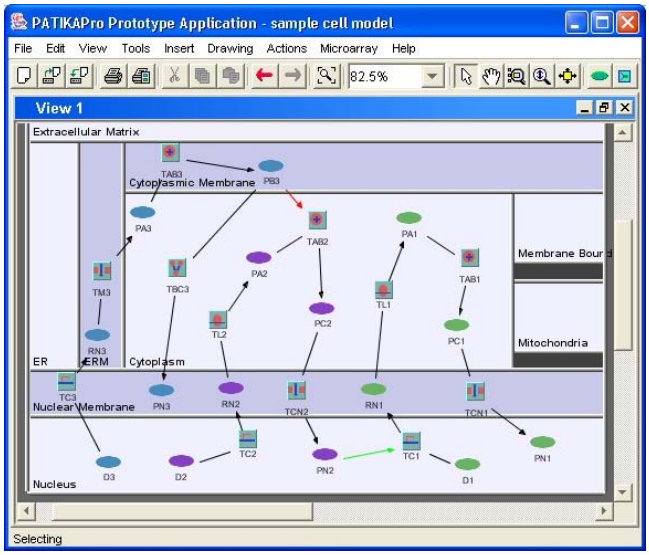

Fig. 5. The cell model assumed by our algorithm and used by PATIKA. Each biological node is confined to its compartment.

3.3 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. 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 6). 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.

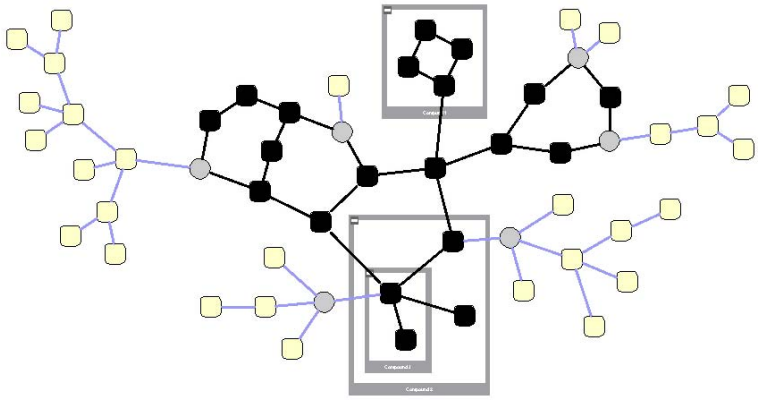

Fig. 6. The *skeleton* is shown dark and reduced trees are marked with light color. Notice that only the trees that are members of the root graph are allowed to be reduced. Reduced tree roots are shown with circle nodes as they will be the tree growth origins for later phases of the algorithm, where 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 ∈ 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 ∈ 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 ∈ V do
2)    center := u.ownerGraph.boundRect
3)    calculate gravitation force F_g towards center
```

```
4)   F_g(u) += 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 ≥ 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) += F_rc
7)   F_s(v) −= 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
5) while phase ≤ 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 ≠ 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()
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.

4. IMPLEMENTATION

The algorithm described above has been tested within the example application of Tom Sawyer Visualization for Java, version 7.0 beta. 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).

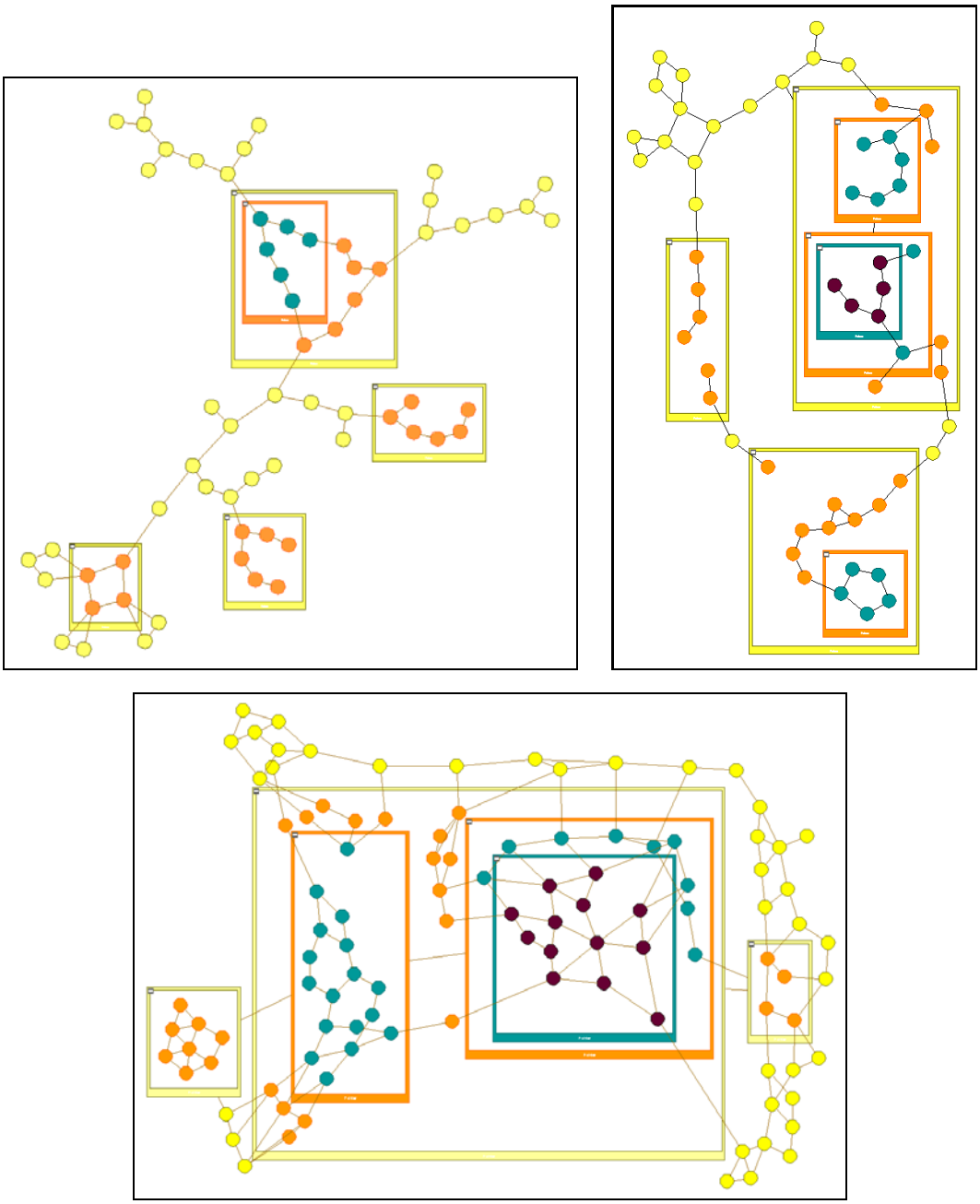

Fig. 7. Sample compound graphs (with varying desired edge lengths and edge and intergraph edge density) laid out by our algorithm. The nodes are color-coded to denote the depth of the node in the nesting hierarchy (i.e., the deeper a node is, the darker its color is).

The results have been found quite satisfactory as far as the general graph drawing criteria such as number

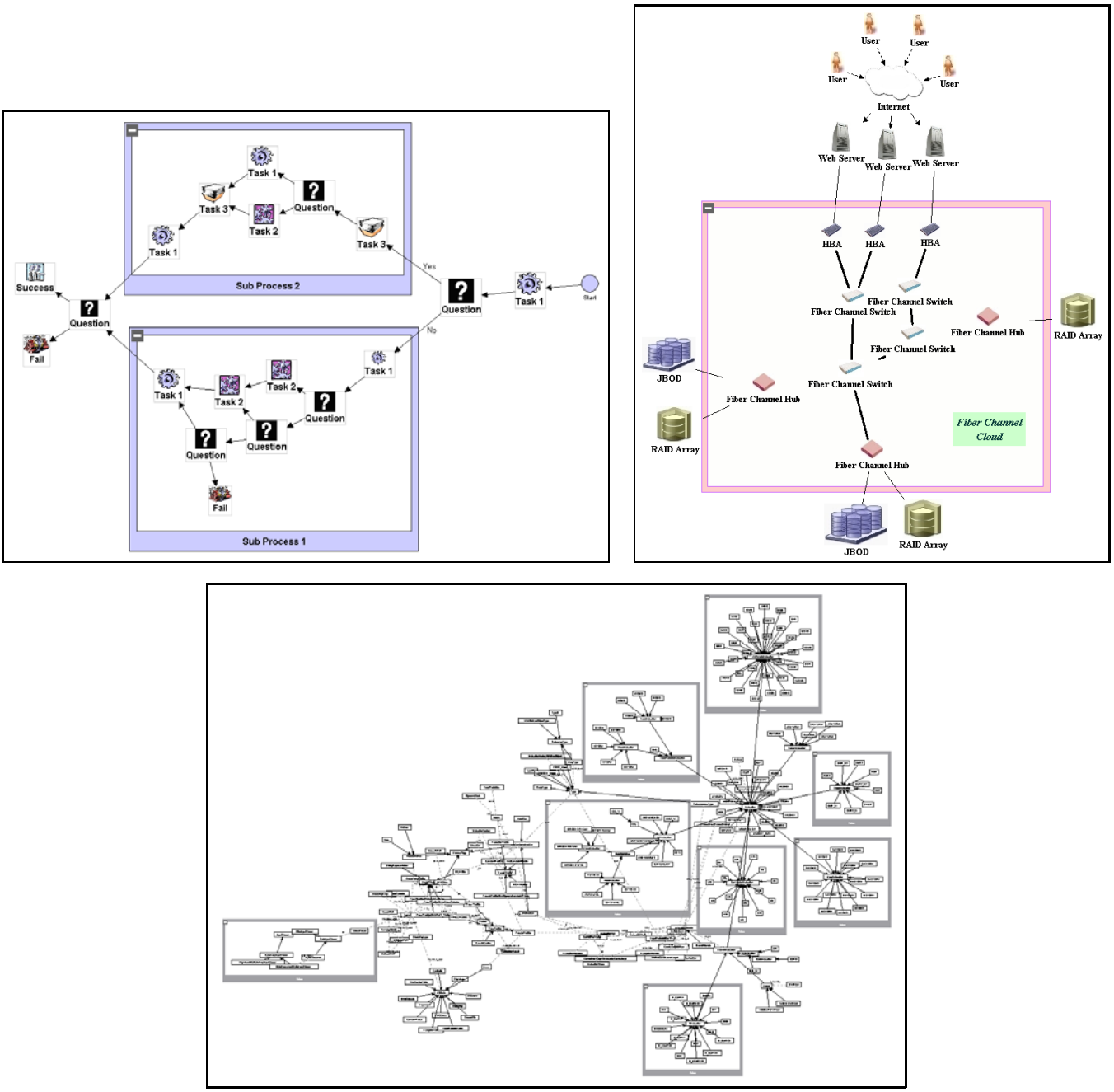

Fig. 8. Sample real-life compound graphs (with varying desired edge lengths and node sizes) from business workflow, networking, and software modeling (courtesy of Tom Sawyer Software), laid out with our algorithm.

of crossings and total area are concerned (Figures 7 and 8). 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.

4.1 Experimental Results

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 9 shows an example of a randomly generated graph.

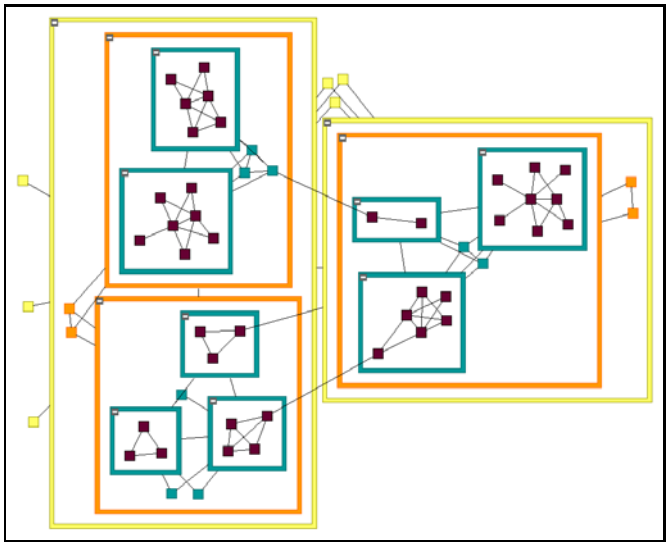

Fig. 9. A randomly generated graph laid out by our algorithm.
($n = 70$, $m/n = 1.5$, $m_{ig}/m = 0.03$, $d = 3$, $b = 3$, and $p = 0.33$)

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 (Figure 10).

We have also experimented with the nesting depth (Figure 11). 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

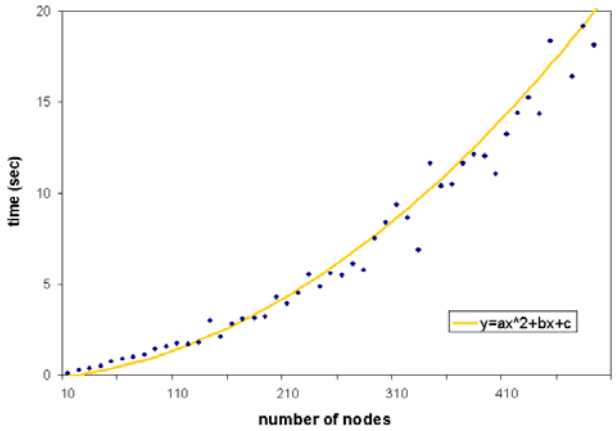

Fig. 10. Number of nodes (n) vs. execution time of our algorithm.
($m/n = 1.5$, $m_{ig}/m = 0.05$, $d = 3$, $b = 3$, and $p = 0.33$)

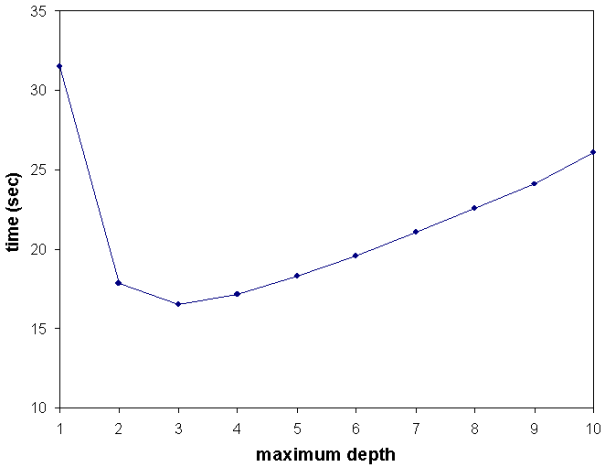

Fig. 11. Maximum nesting depth (d) vs. execution time of our algorithm.
($n = 500$, $m/n = 1.5$, $m_{ig}/m = 0.05$, $b = 3$, and $p = 0.33$)

as the nesting depth increases the performance decreases dramatically due to the increase in the number of compound nodes and nested graphs.

Furthermore, we have performed a test set to see how the proportion of intergraph edges to regular edges affect the execution time (Figure 12). As expected the time it takes to process an intergraph edge as opposed to a regular edge does not vary much.

Lastly we wanted to see how the average number of nested graphs per graph affected the execution time (Figure 13). 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.

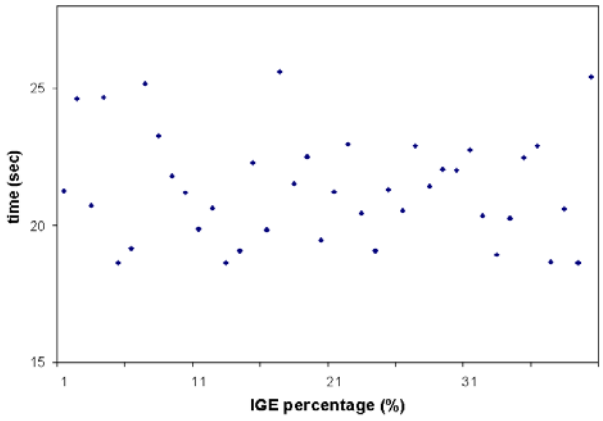

Fig. 12. Proportion of intergraph edges to all edges (m_{ig}/m) vs. execution time of our algorithm. ($n = 500$, $m/n = 1.5$, $d = 3$, $b = 3$, and $p = 0.33$)

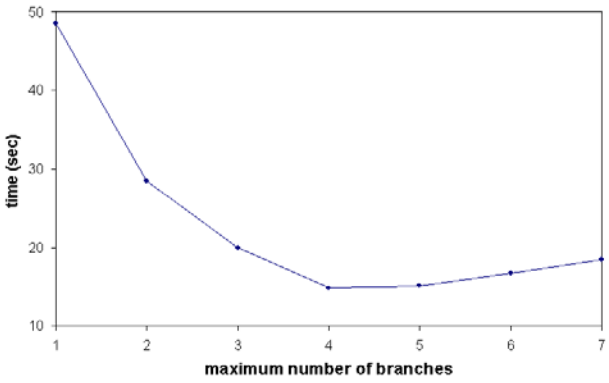

Fig. 13. Maximum branching in the nesting tree (b) vs. execution time of our algorithm. ($n = 500$, $m/n = 1.5$, $m_{ig}/m = 0.05$, $d = 3$, and $p = 0.33$)

4.2 An Application

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

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 14 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.

4.3 Implementation Issues

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

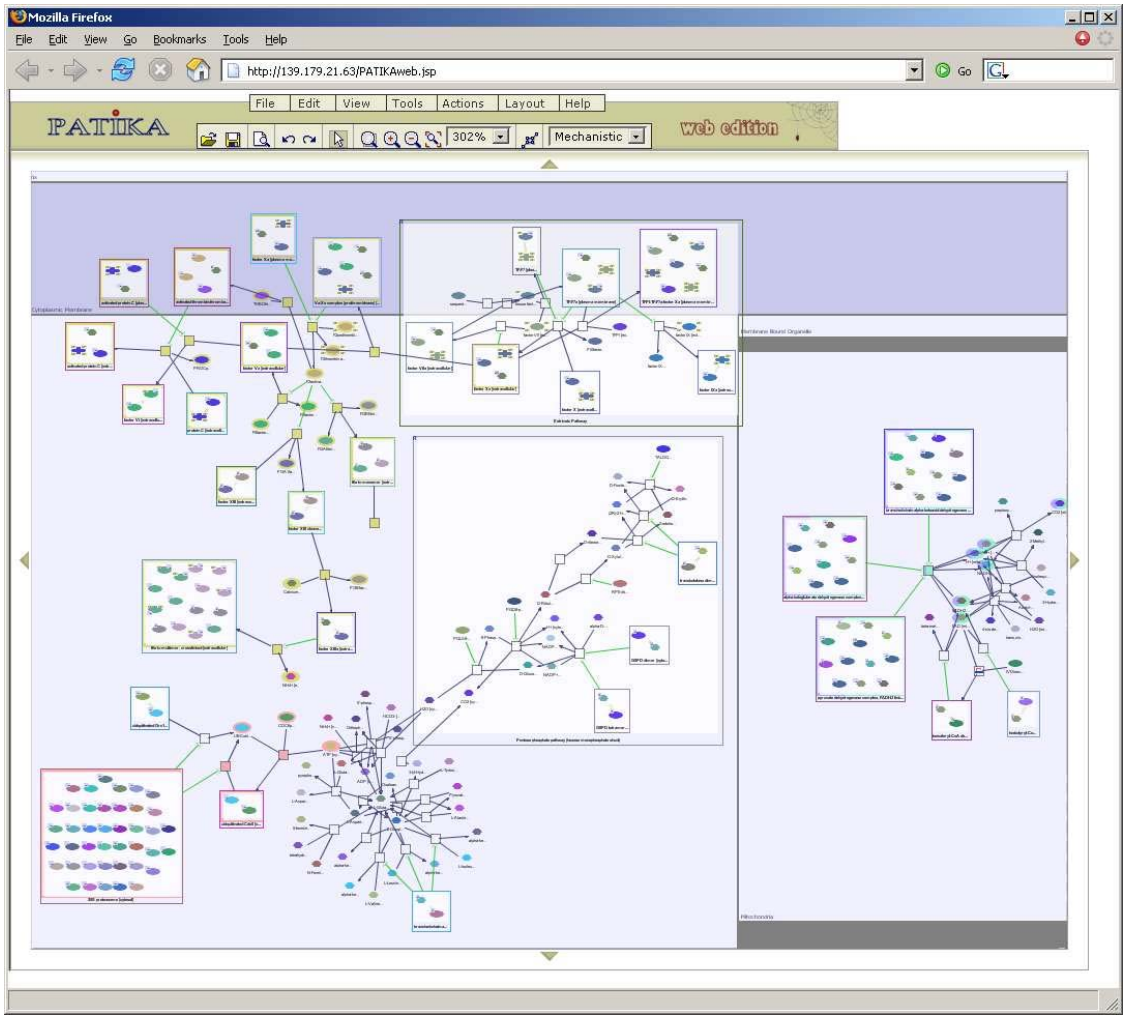

Fig. 14. A sample pathway (obtained by a query from the PATIKA database) laid out by our algorithm.

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 [Frick et al. 1995] 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.

5. 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 novelties of our work include the use of a modified spring embedder system that treats compound nodes and intergraph edges as part of the 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.

ACKNOWLEDGMENTS

The authors would like to thank Tom Sawyer Software, Oakland, CA for providing us with their graph visualization software.

REFERENCES

BERTAULT, F. AND MILLER, M. 1999. An algorithm for drawing compound graphs. In *Graph Drawing (Proc. GD '99)*, J. Kratochvil, Ed. Lecture Notes in Computer Science, vol. 1731. Springer-Verlag, 197–204.

BOHRINGER, K. AND PAULISCH, F. 1990. Using constraints to achieve stability in automatic graph layout algorithms. In *CHI '90 Proceedings*. ACM, 43–51.

DEMIR, E., BABUR, O., DOGRUSOZ, U., GURSOY, A., NISANCI, G., CETIN-ATALAY, R., AND OZTURK, M. 2002. PATIKA: An integrated visual environment for collaborative construction and analysis of cellular pathways. *Bioinformatics 18,* 7, 996–1003.

DI BATTISTA, G., EADES, P., TAMASSIA, R., AND TOLLIS, I. G. 1999. *Graph Drawing, Algorithms for the Visualization of Graphs*. Prentice-Hall.

DOGRUSOZ, U., FENG, Q., MADDEN, B., DOORLEY, M., AND FRICK, A. 2002. Graph visualization toolkits. *IEEE Computer Graphics and Applications 22,* 1 (January/February), 30–37.

EADES, P. 1984. A heuristic for graph drawing. *Congressus Numerantium 42*, 149–160.

EADES, P., FENG, Q., AND LIN, X. 1997. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In *GD '96*, S. North, Ed. Lecture Notes in Computer Science, vol. 1190. Springer-Verlag, 113–128.

FRICK, A., LUDWIG, A., AND MEHLDAU, H. 1995. A fast adaptive layout algorithm for undirected graphs. In *GD '94*, R. Tamassia and I. Tollis, Eds. Lecture Notes in Computer Science, vol. 894. Springer-Verlag, 388–403.

FRISHMAN, Y. AND TAL, A. 2004. Dynamic drawing of clustered graphs. In *Proceedings of IEEE Symposium on Information Visualization*. 191–198.

FRUCHTERMAN, T. M. J. AND REINGOLD, E. M. 1991. Graph drawing by force-directed placement. *Software Practice and Experience 21,* 11, 1129–1164.

FUKUDA, K. AND TAKAGI, T. 2001. Knowledge representation of signal transduction pathways. *Bioinformatics 17,* 9, 829–837.

K. SUGIYAMA AND K. MISUE. 1995. A Generic Compound Graph Visualizer/Manipulator: D-ABDUCTOR. In *GD '95*, F. J. Brandenburg, Ed. Lecture Notes in Computer Science, vol. 1027. Springer-Verlag, 500–503.

LAI, W. AND EADES, P. 1996. A graph model which supports flexible layout functions. Tech. Rep. 96-15, Callaghan 2308, Australia.

RAITNER, M. 2002. HGV: A library for hierarchies, graphs, and views. In *Proc. Graph Drawing '02*, M. Goodrich and S. Kobourov, Eds. LNCS, vol. 1528. 236–243.

SANDER, G. 1996. Layout of compound directed graphs. Tech. Rep. A/03/96, University of Saarlandes, CS Dept., Saarbrücken, Germany.

SUGIYAMA, K. AND MISUE, K. 1991. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man and Cybernetics 21,* 4, 876–892.

Wang, X. and Miyamoto, I. 1995. Generating customized layouts. In *Graph Drawing (Proc. GD '95)*, F. Brandenburg, Ed. Lecture Notes in Computer Science, vol. 1027. Springer-Verlag, 504–515.