USING CONSTRAINTS TO ACHIEVE STABILITY IN AUTOMATIC GRAPH LAYOUT ALGORITHMS

*Karl-Friedrich Böhringer**　　　*Frances Newbery Paulisch*

Institute for Informatics
University of Karlsruhe
D-7500 Karlsruhe 1
West Germany
+49–721–6084068, newbery@ira.uka.de

ABSTRACT

Automatic layout algorithms are commonly used when displaying graphs on the screen because they provide a "nice" drawing of the graph without user intervention. There are, however, a couple of disadvantages to automatic layout. Without user intervention, an automatic layout algorithm is only capable of producing an aesthetically pleasing drawing of the graph. User- or application-specified layout constraints (often concerning the semantics of a graph) are difficult or impossible to specify. A second problem is that automatic layout algorithms seldom make use of information in the current layout when calculating the new layout. This can also be frustrating to the user because whenever a new layout is done, the user's orientation in the graph is lost.

This paper suggests using layout constraints to solve both of these problems. We show how user-specified layout constraints may be easily added to many automatic graph layout algorithms. Additionally, the constraints specified by the current layout are used when calculating the new layout to achieve a more stable layout. This approach allows a continuum between manual and automatic layout by allowing the user to specify how stable the graph's layout should be.

KEYWORDS: Graphical user interfaces, graph layout algorithms, layout constraints.

*Current address: Dept. of Computer Science, 4107 Upson Hall, Cornell University, Ithaca, NY 14853, karl@cs.cornell.edu

1 INTRODUCTION

Graphs, consisting of a set of nodes and a set of edges, are one of the most fundamental ways of representing relationships among objects. Programs that display a set of relationships as a graph [12, 11, 14, 2, 4] have become more prevalent in recent years because of two major factors. Firstly, a person is usually able to comprehend information better when it is presented pictorially (for example a graph) rather than in textual form. This is partly due to the fact that structural properties such as planarity, symmetry, and hierarchy are readily apparent from a well drawn graph and recognition of these properties seems to help the user "understand" the graph. Secondly, the proliferation of high quality graphics workstations has made the use of graphs as a significant part of the graphical user interface affordable and available to many users.

1.1 Definitions

This subsection will provide some definitions that will be used throughout the paper. The position of each node and edge in the graph is called the *layout* of the graph. Graph layout can either be done *manually*, meaning that each node and edge is placed by the user, or *automatically*, meaning that an algorithm computes the position of the nodes and edges to produce a "nice" layout. What constitutes a "nice" layout depends on the type of graph, the application, as well as on the user's taste. Typically, an automatic graph layout algorithm tries to meet one or more (possibly conflicting) aesthetic goals. Minimizing the number of edge crossings, maximizing the symmetries, or minimizing the total area of the graph are some of the many possible aesthetic goals.

Automatic layout has the advantage of relieving the user of the tedious chore of layout, but usually does not produce quite as good results as a manual layout. One of the main reasons is that most automatic layout algorithms are not designed to take a user's *layout*

Proc. of ACM SIGCHI Conf. on Human Factors in Computing Systems, Seattle, WA, April 1990.

constraints into account. For example, the user might request that one node be to the left of another or that a particular group of nodes be placed near each other. Since these constraints are specified explicitly, they are of higher priority than the aesthetic goals of the layout algorithm.

A graph whose layout does not change much when it is newly layed out is called *stable*. *Structural stability* is concerned with meeting the user-specified layout constraints. If many user-specified layout constraints are specified and satisfied then the graph will not have much freedom of movement. *Dynamic stability* is concerned with minimizing the difference between successive layouts of the graph [15]. Messinger [6] suggests that the difference be measured as "how many and how far vertices and edges move from their previous locations". Ideally, making a minor change in the graph's structure should cause only a minor change in the layout. Most automatic layout algorithms do a complete new layout without taking the current layout into account at all. This implies that the new graph layout may be dramatically different from the previous one. This can be very frustrating to the user because they lose their orientation in the graph.

1.2 Overview Of Our Solution

This paper will describe a general mechanism for extending automatic layout algorithms to be able to handle layout constraints. One particular layout algorithm is used throughout as an example, but the same approach would work on many other layout algorithms as well.

Our solution [3] uses layout constraints to achieve both structural and dynamic stability. As can be seen from the overview of our solution (Figure 1), our approach is layered. Constraints for a single dimension are specified at the lowest level. The next higher level manages those constraints. On top of this comes the 3-D constraint manager which combines the constraints from the three dimensions and provides a common interface to the graph layout algorithm used by the application.

The following section will describe how the constraints are represented and how the possibly conflicting constraints are evaluated. Section Three describes how a layout algorithm can be extended with a constraint manager to achieve structural and dynamic stability. Section Four briefly describes how we have integrated this solution into the EDGE graph editor [8, 16]. Section Five gives several examples that demonstrate how using layout constraints contribute to layout stability. Finally, Section Six summarizes our work and suggests some future directions.

2 REPRESENTATION AND EVALUATION OF CONSTRAINTS

In this section we describe the constraints used in our system and the way they are managed. Following the architecture, we start with simple one-dimensional constraints and end up with consistent three-dimensional constraint networks.

2.1 Low Level Constraints

First let us describe the kind of constraints that a user would typically like to have available. To constrain the position of a node in a graph, there are three different types of constraints:

Absolute Positioning: Constrain the node's position in regards to a fixed coordinate system. For example, assuming that nodes are placed in horizontal levels, constrain placement of a node to a particular level ("level 4") or to a range of positions within a particular level ("level 2, position 3–5").

Relative Positioning: Constrain the node's position in relation to other nodes. For example, "node A is left of node B" or "node C is the top neighbor of node D".

Clusters: Gather a group of nodes together to a "cluster" which can then be further constrained. For example, "cluster E must have a maximum width of 3 units" or "all nodes in cluster F are to the right of node G".

To describe these constraints, we introduce a coordinate system. The x-axis runs from left of right, the y-axis from top to bottom. The origin of the coordinate system is assumed to be in the upper left corner. For three-dimensional layout there may also be a z-axis running from the front to the back.

The constraints can be formulated using the coordinates of each node. For example "node A is vertically above node B" is described by the two equations $A.x = B.x$ and $A.y < B.y$. This example reveals two principles our system is based on:

1. Different dimensions are treated independently from each other.

2. Constraints are restricted to linear equations.

We have found that these two principles pose no severe restriction on the layout constraints we can define. Note that even constraints like "node A is above and to the left of node B" can be described by the two independent equations $A.x < B.x$ and $A.y < B.y$. The main restriction due to linear equations is the

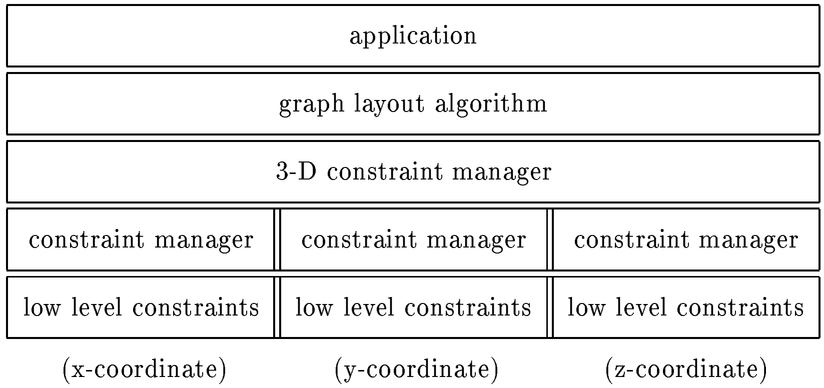

Figure 1: Overview of System Architecture

way distances are measured (Manhattan metric as opposed to Euclid metric [1]) and the lack of transcendent or trigonometric functions.

On the other hand, principle 1 makes implementation much simpler (two- or three-dimensional constraints are no more difficult than one-dimensional constraints). Principle 2 is crucial for an efficient evaluation of the constraints. The evaluation of sets of linear equations is an common problem, for instance in temporal data bases [1, 10].

In conclusion of this section we define a *low level* (one-dimensional) *constraint* as a linear equation of two variables. We call any set of these constraints a *constraint network*.

2.2 Constraint Manager

For each dimension, there is a constraint manager which has two main tasks:

- Maintain a list of all constraints and provide functions to add, delete, and query the status of constraints in the constraint network.
- Evaluate the constraint network and keep it consistent. A set of constraints is defined to be *consistent* if none of them are contradictory.

The purpose of the evaluation of the constraint network (usually called "constraint propagation" [7, 5]) is to compute the global effects of local constraints. For example, from a chain of order relations like $A.x < B.x$, $B.x < C.x$, $C.x < D.x$ the relation $A.x < D.x$ should be derived. This evaluation can be done in linear time in the number of constraints by an algorithm based on topological sorting. After this preprocessing step, queries can be answered in constant time. For example, the system would answer a query "$A.x < D.x$?" with "TRUE" or "$B.x = D.x$?" with "FALSE", each in $O(1)$ time. This efficiency is important for layout algorithms which may make extensive computations while reordering nodes in the graph layout.

In the case where the constraints are not consistent (i.e. there are contradictory constraints), the constraint manager computes a subset of consistent constraints by deactivating some of the constraints. The selection of deactivated constraints can be influenced by assigning priorities to them. The constraint manager then tries to keep high priority constraints active, while some low priority constraints are deactivated. Among inconsistent constraints with equal priorities, the selection is arbitrary. Deactivated constraints are ignored during the evaluation of the constraint network.

In our implementation the detection of which constraints are deactivated is done by a binary search through inconsistent sets of constraints until all constraints causing an inconsistency are deactivated. This solution, however, increases the time complexity from $O(n)$ to $O(n^2 \log n)$ (where n is the number of constraints) in the worst case and this leaves some room for improvement.

2.3 Three-dimensional Constraints

So far the dimensions have been treated independently. However, in order to define a convenient interface to the user, to the layout algorithms and to the applications programs an interfacing module is used. This module provides functions to translate three-dimensional constraints into one-dimensional ones using the constraint managers for each dimension. Each of these functions can be invoked in three ways:

DO: Insert a new constraint.

[1] Euclid: $d(x,y) = \sqrt{\sum (x_i - y_i)^2}$
Manhattan: $d(x,y) = \sum |x_i - y_i|$

UNDO: Delete an old constraint.

QUERY: Test whether a constraint is met.

3 INTEGRATION WITH AUTOMATIC LAYOUT ALGORITHMS

In this section we want to show how layout constraints may be integrated into an automatic layout algorithm. As we stated before, the constraints are designed to meet user requests rather than the aesthetic goals of a particular layout algorithm. Therefore, our system should be adaptable to several different ones. In the following we describe the integration into Sugiyama's layout algorithm [13]. This layout algorithm or some variation thereof is used in several systems that display directed graphs [12, 6, 4]. First we show how structural constraints are taken into account, then we use this mechanism to achieve dynamic stability.

3.1 Structural Stability

The following is a description of Sugiyama's layout algorithm, which is divided into four phases:

Topological Sorting: Assign nodes to levels according to their depth (longest path of predecessors) in the graph. Cycles in the graph are handled by temporarily reversing the direction of an edge.

Subdivision Of Long-span Edges: Split "long" edges that span more than one level into a series of shorter ones by inserting "dummy" nodes at all in-between levels.

Barycentric Ordering: Determine the relative positions of nodes within each level where the goal is to reduce the crossings with the adjacent level. Each node is positioned based on its *barycenter* which, roughly speaking, is the average position of its predecessors (or successors). Several upward and downward passes are made through the graph until no improvement is detected or a threshold value has been reached.

Finetuning: Determines the actual x, y coordinates of each node. The finetuning shifts nodes within their level to center nodes in respect to their predecessors/successors. The relative position of the nodes is not allowed to change, so this phase will not contribute to any more (or less) edge crossings.

For each of these phases some changes or extensions to the original algorithm were necessary. Before doing so we have to define the correspondence between the coordinate system used by the constraints and the layout algorithm. In x-direction coordinate units correspond to subsequent positions. In y-direction the levels are assigned subsequent numbers. Together a constraint "A is the left neighbor of B" can be defined by the two equations "$A.x + 1 = B.x$" and "$A.y = B.y$".

Integration of constraints into the first phase is easy because constraint evaluation includes topological sorting. For each edge, the algorithm introduces one constraint stating that the source node should be placed above the target node of the edge. These automatically generated constraints receive a priority that is lower than user-specified ones. Evaluation of the constraints then yields a proper level assignment. Due to user-specified constraints, additional back edges (source is below target) and also "flat" edges (running between nodes on the same level) may arise. Back edges are temporarily reversed like edges forming a cycle. But as Sugiyama's algorithm is not designed to handle "flat" edges (it would draw an edge straight through all intermediate nodes), an additional constraint is generated which requires that these nodes are immediate neighbors.

The second phase is also easy to adapt. If there are two nodes constrained to lie in the same vertical line it is reasonable to require that an edge between them also runs straight on this line rather than being allowed to bend. Therefore additional constraints are generated to force all intermediate nodes to have the same x-coordinate.

The main work of the Sugiyama layout is done during the third phase when a total ordering of the nodes in each level is determined. After computing the barycentric ordering of the nodes (resulting for example in $A \rightarrow B \rightarrow C$), corresponding constraints ($A_x < B_x, B_x < C_x$) are given to the constraint manager. They receive a low priority so that in determining the final ordering, the constraint manager will give preference to user- or application-specified constraints. It is important that the constraint manager be as efficient as possible (in our case $O(1)$ for queries) since there is a large number of constraints and because the levels are rearranged frequently.

Minimization of edge crossings only makes sense in two-dimensional space. If we use a third dimension the resulting graph layout remains a two-dimensional projection. This means that we have to minimize edge crossings in the projection. Therefore we do not introduce a third dimension until the final finetuning phase. For nodes constrained to lie in front of each other we rather generate internal constraints which request that these nodes are immediate neighbors on the same level. During the last phase we can adjust them into the intended $2\frac{1}{2}$-D position [2].

[2] $2\frac{1}{2}$-D means that the nodes will overlap slightly, like a spread deck of cards.

In the final phase (finetuning) the relative ordering of the nodes is preserved, but the x-coordinates of the nodes are determined in a level-by-level pass through the graph. When determining the position of nodes in the current level, the position of nodes in the previous levels must be taken into account. Additionally, if three-dimensional layout is used, nodes which are constrained to lie in front of each other are positioned slightly overlapping to achieve the $2\frac{1}{2}$-D effect.

3.2 Dynamic Stability

The previous section described how Sugiyama's algorithm could be adapted to handle structural stability. Now we will use the same constraint mechanism to achieve dynamic stability. Although it is generally agreed that dynamic stability is a serious problem with automatic layout algorithms, dynamic stability is still a relatively unexplored research area. Most approaches try some form of "incremental layout" meaning that only a small portion of the graph is newly laid out whereas the rest of the graph remains constant. This is particularly important for very large graphs where the extensive computations of the layout algorithm may consume a considerable time.

Our approach is to generate additional constraints after each automatic layout. If the graph is edited these constraints will be weakened in the vicinity of changes. This causes the graph layout to be flexible in changed areas while it remains stable in the remainder of the graph.

The *vicinity* of a change is a subgraph close to where the change occurred. It includes the node(s) directly affected by the change plus nodes that are some number of edge length away from the directly affected nodes. The number of edge lengths is a user-specified parameter that describes the degree in which the layout should change. This may vary from extremely stable (vicinity contains only the directly affected nodes) to unconstrained (vicinity contains all nodes of the graph, therefore same results as standard Sugiyama).

The constraints generated to achieve dynamic stability constrain each node to its level in the current layout and connect nodes within a level into chains representing their order. After a change has occurred, flexible nodes are freed from these constraints. During the new layout they can move freely, while the other nodes keep the relative positions they had in the old layout. If all nodes on a level remain stable, their order need not be recomputed, thus speeding up the layout.

From our experience, the large number of generated constraints degrades the performance such that their is no significant gain in performance as compared to the original layout algorithm. The performance, however, does not suffer too badly because the number of generated constraints is linear in the number of nodes and no inconsistencies are introduced. Thus using this method results in a stable graph layout at roughly the same speed as a completely new layout, producing a trade-off between Sugiyama's aesthetics and dynamic stability.

4 INTEGRATION WITH GRAPH EDITOR

The EDGE graph editor [8, 16], which offers a choice of several layout algorithms for displaying and editing graphs, was extended to include the modified Sugiyama algorithm and the associated constraint manager. The following three subsections briefly describe the modifications made to the user, input/output, and application interfaces.

4.1 User Interface

The user specifies editing operations via a set of pop-up menus. We extended the set of menus so that the user can list or alter the current list of constraints. To alter a constraint, the user selects the list of nodes by clicking them with the mouse, fills out a form-like menu specifying the type of constraint, the priority etc., and then selects "DO", "UNDO" or "QUERY". The appropriate command is then sent to the 3-D constraint manager which responds accordingly.

4.2 Input/Output Interface

The default input/output format is a high-level textual description of the format and appearance of the nodes and edges in the graph called GRL (Graph Representation Language) [9]. The portion of the GRL describing the constraints and their attributes is delimited by keywords and each constraint is a set of *attribute:value* pairs. The following is an excerpt of a GRL description specifying that node A should be to the left of node B (with a priority value of 10) and that node B should be in the same vertical column as node C (with default priority 0).

```
constraint:
    constraint: left
        nodes:("A","B")    priority: 10,
    constraint: equal_column
        nodes:("B","C")
endconstraint:
```

4.3 Application Interface

EDGE can be customized to many graph-based applications (for example PERT charts for project man-

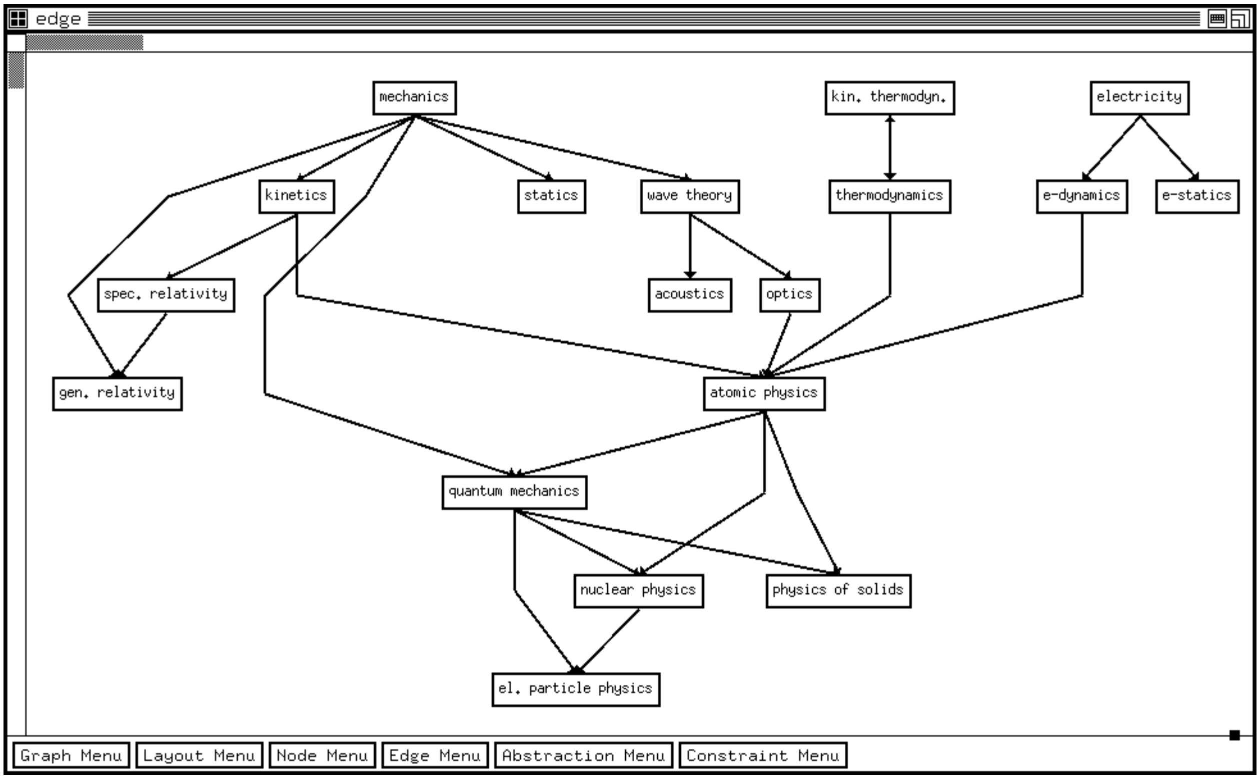

Figure 2: Interdependencies in Physics

agement, call graph animation, or directory browsing). The applications may invoke any of the functions offered by the 3-D constraint manager to specify application-specific constraints. For example, our PERT chart graph editor, which calculates the critical path of a project and highlights those nodes, uses the "equal column" constraint to align all of the nodes in the critical path.

5 EXAMPLES

This section is intended to demonstrate the capabilities of the system described above. To begin with, we show an example for structural stability. Figure 2 depicts an overview of the interdependencies between areas of physics. Sugiyama's algorithm is appropriate for the almost-hierarchical structure of the graph. However, the positioning remains somewhat arbitrary because the layout algorithm lacks knowledge of the semantics of the graph. For example, the user would prefer "statics" and "kinetics" as immediate neighbors, "mechanics", "wave theory", "optics" and "atomic physics" on the same x position and "e-statics" in front of "e-dynamics". This is achieved by introducing these constraints either interactively, in the GRL file, or from the application. The result is shown in Figure 3.

To demonstrate dynamic stability let us go back to Figure 2 [3]. If we add an edge from "e-statics" to "nuclear physics", then Sugiyama's algorithm will yield a graph layout like the one in Figure 4. We recognize that small changes in the graph structure may cause dramatic changes in the layout. Figure 5 shows the same graph, but this time making use of dynamic stability. The shape of the graph remains quite similar to Figure 2, making it easier for the user to keep his orientation. Our approach tries to find a compromise between dynamic stability and other layout aesthetics such as the number of edge crossings. Nodes in the vicinity of a change may alter their position. In this example we used vicinity size 1, i.e. nodes involved in a change ("e-statics", "nuclear physics") and their immediate neighbors ("electricity", "atomic physics", "quantum mechanics", "el. particle physics") were allowed to alter their position.

The time to compute the graph layouts for Figures 2 – 5 lay between 3 and 4 seconds, measured as real time on a Sun-3/110 with 8 MByte main memory. This suggests that the additional time involved for structural and dynamic layout is reasonable.

[3] We use Figure 2 rather than Figure 3 as our starting-point because at this point we are concerned only about the dynamic (in)stability in Sugiyama's algorithm. Structural stability would help to overcome this problem, as well.

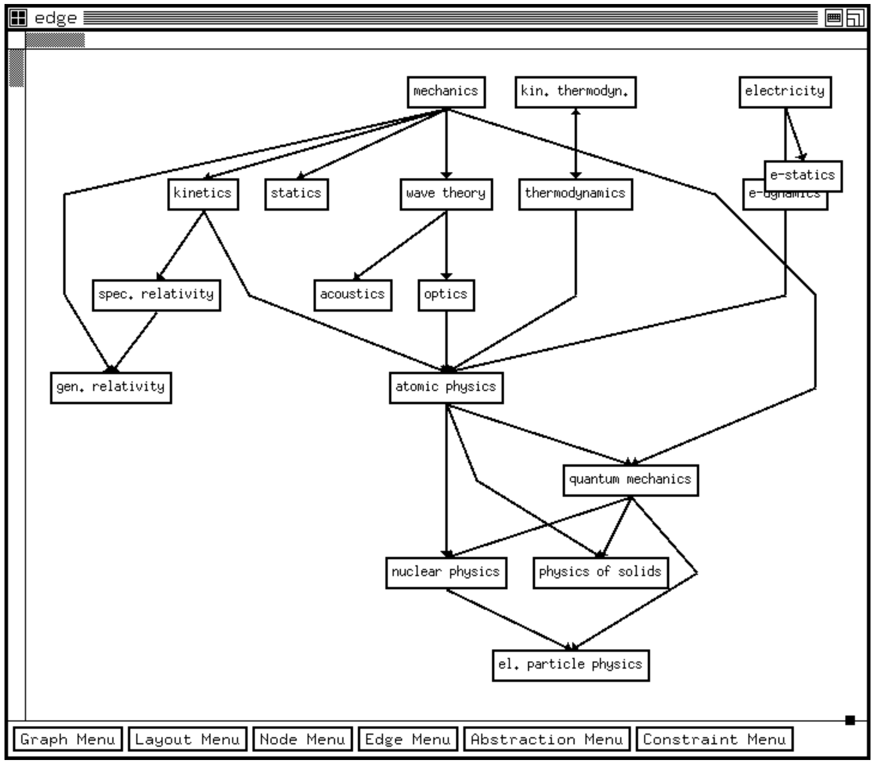

Figure 3: Constrained Layout

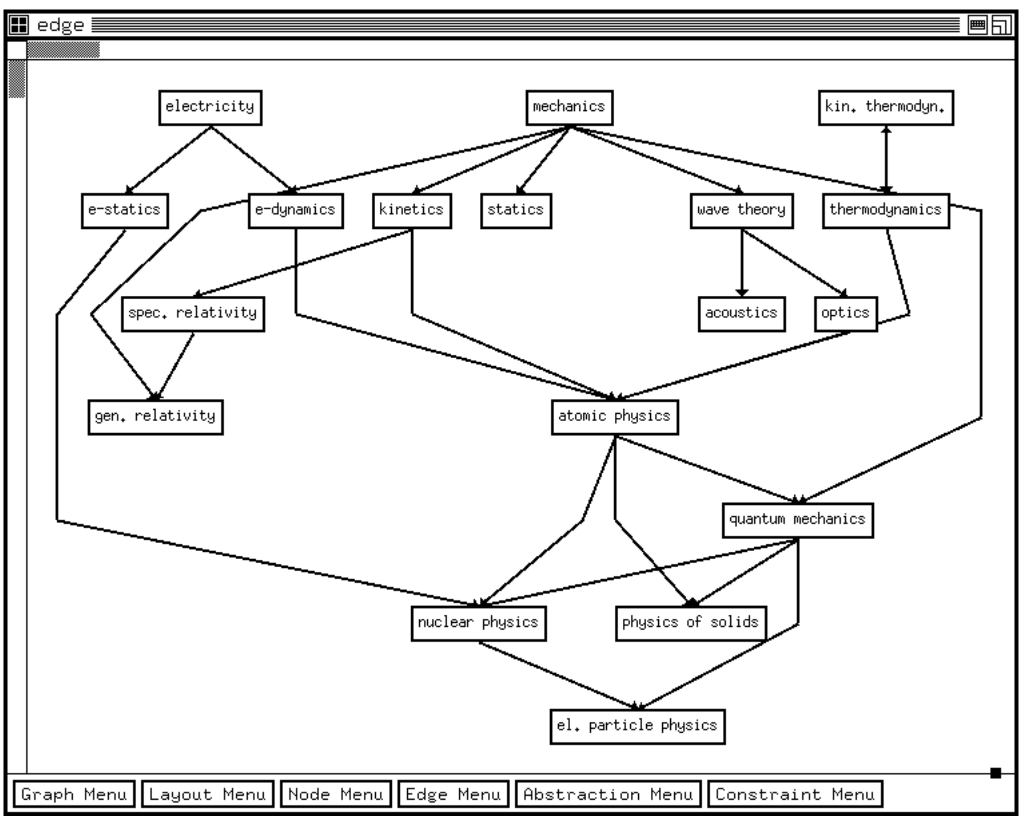

Figure 4: Instablility (edge from "e-statics" to "nuclear physics" added)

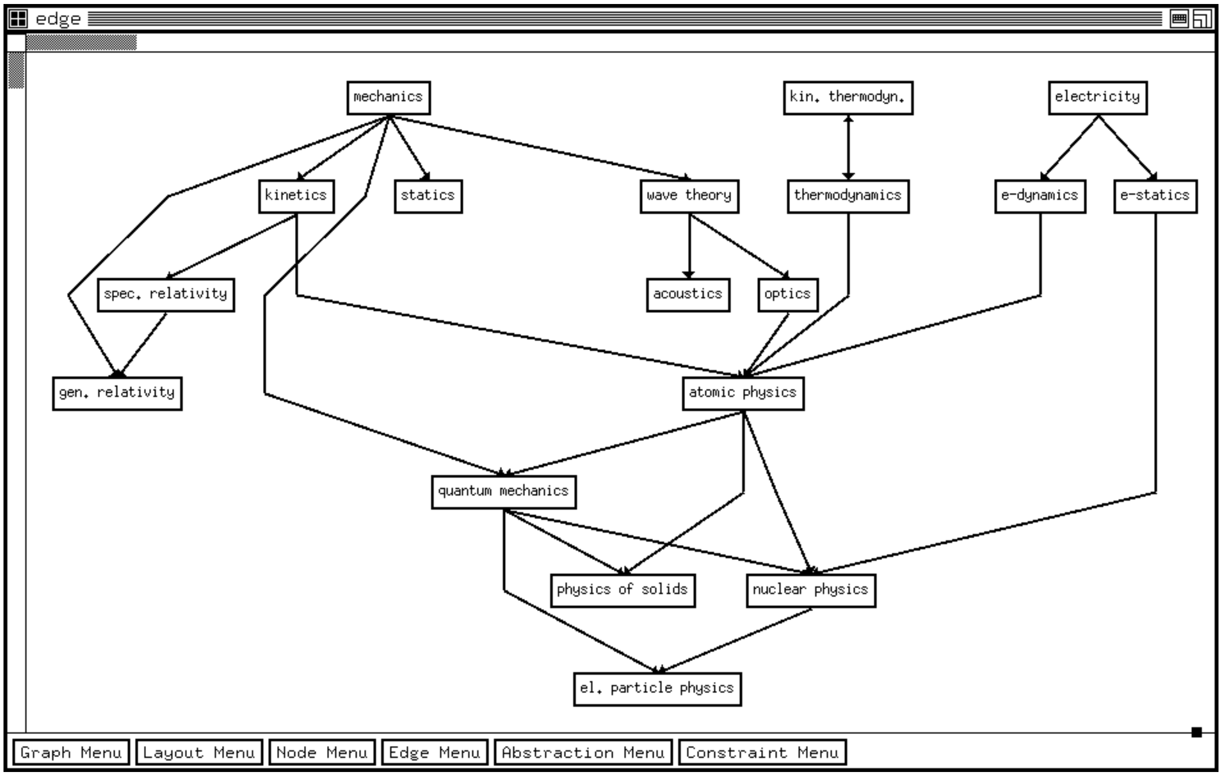

Figure 5: Dynamic Stability

6 RESULTS AND CONCLUSIONS

In this article we proposed a way to achieve stability in automatic graph layout. To this end we defined two types of stability, structural and dynamic. Structural stability deals with constraints on the graph layout imposed by the user or by applications programs. Dynamic stability is the effort to keep subsequent layouts of graphs similar after the graph's structure was changed.

The system described above achieves both types of stability by the same mechanism: layout constraints. The problem was divided into two parts: The representation of constraints, and their integration into an automatic layout algorithm. This division makes it possible to use the constraint representation in various layout algorithms. Representation provides the management of any set of linear equations between two scalar values, in particular components of the node coordinates. This seems to be a reasonable compromise between expressiveness of the constraint language and implementation and efficiency issues. Integration has to be done individually for different layout algorithms, but the necessary changes are straightforward.

This approach allows a continuum between manual and automatic layout because the user can use either structural layout constraints or dynamic stability to affect the stability of the layout. This approach allows application-specific information to be incorporated while retaining the advantages of automatic layout. The user is able to choose a desired degree of dynamic stability by specifying a parameter representing the size of the instable region.

Although the system is definitely useful as it is, there is, of course, still room for improvement and extensions. In particular, the efficiency of the constraint manager when encountering inconsistent constraints could be improved. Application to non-cartesian coordinate systems is also worth investigation. Because constraints are restricted to linear equations, for example in polar coordinates it is not possible to define constraints like "A left of B" without trigonometric functions. On the other hand, this might not be a severe disadvantage, as the circular symmetry of polar coordinates rather implies constraints like "A and B lie in the same sector" or "A is near to the center" which can be easily expressed with linear constraints on the polar coordinates. Other improvements would be a pattern matching mechanism in the "undo" command (for example "undo all constraints involving nodes A and B") or a more sophisticated edge routing in $2\frac{1}{2}$-D layout.

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.

[2] C. Batini, E. Nardelli, and R. Tamassia. A layout algorithm for data flow diagrams. *IEEE Transactions on Software Engineering*, SE-12(4):538–546, April 1986.

[3] K.-F. Böhringer. Stability in graph layout algorithms. Master's thesis, University of Karlsruhe, Institute for Informatics, July 1989. In German.

[4] E. Gansner, S. C. North, and K. P. Vo. DAG: A program that draws directed graphs. *Software—Practice and Experience*, 18(11):1047–1062, November 1988.

[5] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[6] E. B. Messinger. *Automatic Layout of Large Directed Graphs.* PhD thesis, University of Washington, Department of Computer Sciences, July 1989. TR Number 88-07-08.

[7] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[8] F. J. Newbery. EDGE: An extendible directed graph editor. Technical Report 8/88, University of Karlsruhe, Institute for Informatics, June 1988.

[9] F. J. Newbery. An interface description language for graph editors. In *Proc. of the IEEE 1988 Workshop on Visual Languages*, Pittsburgh, PA, October 1988.

[10] F. Puppe. *Introduction to Expert Systems.* Springer Verlag, 1988. In German.

[11] G. Robins. The ISI grapher: a portable tool for displaying graphs pictorially. Computers in Symbolic Graphs and Communications (see. Sven Moer), Helsinki, Finland, August 17-18 1987. Symboliikka '87. Information Sciences Institute, Marina Del Rey, CA.

[12] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Software—Practice and Experience*, 17(1):61–76, January 1987.

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

[14] R. Tamassia, C. Batini, and M. Talamo. An algorithm for automatic layout of entity relationship diagrams. In C. Davis, S. Jajodia, P. Ng, and R. Yeh, editors, *Entity-Relationship Approach to Software Engineering*, pages 421–439. North-Holland Publishing Co, 1983.

[15] R. Tamassia, G. D. Battista, and C. Batini. Automatic graph drawing and readability of diagraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, SE-18(1):61–79, Jan/Feb 1989.

[16] W. F. Tichy and F. J. Newbery. Knowledge-based editors for directed graphs. In H. K. Nichols and D. Simpson, editors, *ESEC'87*, Lecture Notes in Computer Science No. 289, pages 101–109. Springer Verlag, 1987. Proc. of the 1st European Software Engineering Conference.