GraphEd: A Graphical Platform for the Implementation of Graph Algorithms (Extended Abstract and Demo)

Michael Himsolt

Universität Passau, 94032 Passau, GERMANY
himsolt@fmi.uni-passau.de

Abstract. GraphEd is an extensible graph editor. Its powerful object oriented user interface supports all operations that are necessary for the convenient construction and manipulation of graphs. Graph grammars can be used as a macro system to create structured graphs.
GraphEd's modular structure and the application interface support the easy integration of algorithm modules which are written in C, or can run external programs. The user may construct graphs interactively, select algorithms from a menu, and view the results of an algorithm directly on screen. Several graph layout algorithms assist the user to tidy graph drawings, and help the programmer to visualize results or debug complex algorithms.
Actual applications range from standard graph algorithms over graph drawing algorithms, algorithm animation and combinatorial algorithms to front ends for circuit design systems.

1 Introduction

The implementation of graph algorithms is a challenge to every programmer. There is a growing number of systems to support this process, which can be roughly divided into two categories. Systems like LEDA [15] and graphbase [13] concentrate on the implementation of a library of graph algorithms. Systems like EDGE [16], DaVinci [7] or GRAB [17] concentrate on graph visualization and editing, and offer only limited support for implementing algorithms.

There is a gap between these two categories. To program and work with graph algorithms, it is necessary to input and output graphs in a convenient way. It is obvious that a text based description like an adjacency list is too difficult to deal with, so a graphical representation is necessary. For example, a user needs to create input graphs in a convenient way, and would like to see the results of an algorithm. A programmer will need help with debugging, and this cannot be done with adjacency lists in a reasonable way. Systems like CABRI [1] or Design [14] are a combination of an editor and a library of algorithms.

The construction of graphs is still a complex process, even with the support of an interactive system. But, many graphs have a more or less regular structure. A macro system can help to make the generation of structured graphs much easier.

However, an editor component alone is still not sufficient for such a system. When an algorithm creates a graph, it does usually not assign coordinates, so this must be done by some algorithm. And, if the user makes changes in the graph, its drawing will become harder to read, so the graph needs to be rearranged. Since this is always a tedious process, any assistance by the computer is appreciated.

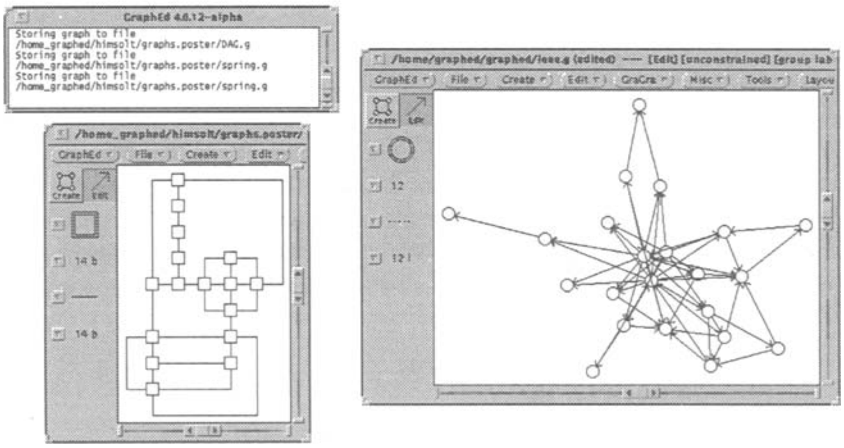
GraphEd [10] is a system which supports all these requirements. GraphEd consists of an editor, an interface for implementing graph algorithms and a set of graph layout algorithms. GraphEd is written in C and runs on SUN workstations under Open Windows or X11.

2 GraphEd

GraphEd is composed of three subsystems, the core system, the application interface and the application modules. The core system is the graph editor itself, which contains the user interface and all basic graph and graph grammar operations. Complex editor functions, algorithms and application specific code are implemented in separate modules. The application interface, which is based on the Sgraph graph data structure, combines both parts.

2.1 The User Interface

Figure 1 shows a screen running GraphEd. Graphs are edited in the working areas. There can be several working areas, and each of them can hold an arbitrary number of graphs. Besides that, there are a message window and many temporary windows for tasks such as selecting files, editing the attributes of nodes and edges or setting parameters of algorithms. Most menus can also be teared off and may stay on the screen as tool bars.



```
Storing graph to file
/home_graphed/himsolt/graphs.poster/DAG.g
Storing graph to file
/home_graphed/himsolt/graphs.poster/spring.g
Storing graph to file
/home_graphed/himsolt/graphs.poster/spring.g
```

GraphEd 4.0.12-alpha
/home_graphed/himsolt/graphs.poster/
GraphEd
File
Create
Edit
14 b
14 b
/home/graphed/graphed/leee.g (edited) ---- [Edit] [unconstrained] [group lab
GraphEd
File
Create
Edit
GraGra
Misc
Tools
Layou
12
12 !

Fig. 1. GraphEd Screen

All operations are menu and mouse based. Each working window has a menu bar, and there are keyboard shortcuts for frequently used commands. We have tried to make the user interface as easy as possible. Typically, it takes a few hours to learn the basic features of GraphEd.

All commands work on the current selection. A selection can be a node, an edge or a subgraph. Most commands are made to work with all types of selections. In general, GraphEd is designed to minimize restrictions to the user. For example, graphs may have unlimited size and degree, and edges may be polylines of any length. Most windows are designed in a way that the user may still modify the graph while the window stays open.

Usually, advanced users will call for additional features that support their particular application. Such features can be added through the application interface.

2.2 Modes

GraphEd has two input modes, "create" and "edit". The first one is used to insert new nodes and edges, while the second one is used modify graphs.

Create Mode Nodes are created with the left mouse button. Edges, including bends, are created with the middle mouse button.

Edit Mode Objects are selected with the left mouse button. The user may select nodes, edges and sets of nodes which form subgraphs. There are also operations that deal with overlapping nodes and edges.
Objects are manipulated with the middle mouse button. The user may move nodes and bends of edges, resize nodes, and add or delete bends. The graphical properties of nodes and edges (see Sect. 3) may be edited in special subwindows.

Both modes are kept as similar as possible. For example, labels may be edited in both modes. Any new object is automatically selected. When a user clicks on a node, GraphEd will always select the node and switch to the edit mode if neccessary.

We have chosen a two-mode approach since a single mode would be too complicated to use. For example, moving a node and creating an edge is performed with a mouse drag operation which starts in a node. In a combined mode, it would be difficult to make a distinction between those two operations. We could use modifier keys or a different mouse operation scheme, but that would be inconvenient and counter-intuitive.

2.3 Graph Grammars

Many text editors and desktop publishing systems offer macros to support common text phrases. Since graphs are far more complex structures than texts, a macro mechanism is needed even more essential. Unfortunately, the substitution mechanism for graphs is more complex than for texts. It is not sufficient to replace a node with a subgraph, since that subgraph would not be connected with the rest graph. There must be a mechanism which creates new edges that connect the inserted subgraph with the rest.

Graph grammars [6] provide such mechanisms. GraphEd supports 1-eNCE graph grammars. Derivation rules are specified in a graphical form [9], and can be applied interactively. 1-eNCE graph grammars are not as powerful as those used in systems like PROGRES [18], but they are easier to use and can be implemented more efficiently. For example, [3] shows how to specify data flow graphs by graph grammars, and draw them with a simple layout algorithm.

3 Graphical Representation of Nodes and Edges

GraphEd offers various options to customize the appearance of nodes and edges:

Node Style Nodes can have arbitrary shapes, either builtin ones or external bitmaps.
Builtin shapes are implemented as C functions. Each such shape consists of three functions: a bitmap drawing function, a Postscript drawing function and a node/edge interface function (see below). External bitmaps are generated by a drawing program. They can be loaded at runtime. However, they may need scaling, which might result in some distortions.

Node geometry Nodes can have arbitrary size, bitmaps will be scaled appropriately. The user may resize nodes interactively.

Edge Style Edges can have one of six different dash styles. Since these are actually implemented as tables, additional styles can easily be added if neccessary.

Edge Line Edges are drawn as polylines. The number of segments is not limited. Bends can be inserted, moved and deleted interactively.

Labels GraphEd supports arbitrary labels for nodes and edges. There is no restriction on the contents or the length of the labels.
Node labels are clipped at the boundary of the node. Edge labels may also be clipped.

Fonts Node and edge labels may have arbitrary fonts.

Label Placement GraphEd places labels automatically. Node labels are placed in one of five locations: at the center, or in one of the corners. Edge labels are automatically placed at the middle of the edge.

Node/Edge Interface GraphEd adjusts edges automatically to their end nodes. There are six options how edges are routed.

The implementation is kept as flexible as possible. We have taken much care to make extensions to the list simple. To add a new alternative to an existing option, it is usually sufficient to add the code for this option and update some tables or enumerations. New options need a bit more work, since the file format (see Sect. 5.2) and the graphics operations (see Sect. 6) must be extended, and new dialog elements must be added to the user interface.

GraphEd can output graphs in Postscript or fig format. The latter one can subsequently be processed by the `xfig` program.

4 Sgraph

Sgraph is the data structure which is used for implementing graph algorithms with GraphEd. Sgraph is based on adjacency lists, and can handle both directed and undirected graphs. It is composed of three data structures, **`Sgraph`**, **`Snode`** and **`Sedge`**. An additional data structure **`Slist`** is available to implement sets and lists.

Unlike many other graph data structures (for example, LEDA [15]), Sgraph uses the same data structure for directed and undirected graphs. The main difference is that nodes in directed graphs have lists of incoming and outgoing edges, whereas nodes in undirected graphs have only one list of edges. This means that many algorithms can be implemented for both variants with low overhead.

Insertions and deletions can be performed in constant time. Since all lists are doubly linked and closed, we can traverse the structures from any starting point in both directions. Also, self loops and multiple edges are supported naturally.

Most data structures in the Sgraph library are parametrized. Sgraph provides a data structure **`Attributes`** to attach user-defined attributes to any data structure. Since global data structures like graphs are shared by many different modules, one single set of attributes is not sufficient. For example, if one module calls another, the attributes could be overridden. Or, if attributes are used for longer than a single instance, for example in a sequence of clicks, then there is no control whether other modules can interfere and overwrite the attributes. To accomplish with that, GraphEd supplies a mechanism to save and restore attributes under a given key.

The core system uses a slightly different graph data structure, which is a variant of Sgraph. It contains additional data structures for managing the graphics and the user interface, and is heavily optimized for this purpose. This data structure implements only directed graphs, because the user interface itself does not need to distinct between directed and undirected graphs.

5 The Application Interface

GraphEd's application interface handles the integration of application modules into GraphEd. Sgraph's data structures provide a common base for all implementations. The key concept is that algorithms may do whatever modifications on the graph structure, and the application interface is sole responsible to carry these modifications over to the core system's structures.

The procedure `call_sgraph_proc(proc)` runs an Sgraph based procedure `proc`:

1. Get the current selection and determine the currently selected graph, if there is any.
2. Run `proc`, supplied with the graph, the selection and other information. `proc` may perform all sorts of changes. However, these changes do not immediately affect the data structures in GraphEd. Especially, GraphEd's graph data structure is not automatically updated.

This behavior can be overwritten, so that GraphEd's data structures are changed immediately, for example to implement algorithm animation.
3. After `proc` is done, check for changes in the graph structure, coordinates and labels, and transfer all changes to the core system structures. `proc` may give indicators to speed up this and the following lookups. For example, a planarity test will not change the graph structure and should indicate "no changes done".
4. Redraw the graph if neccessary, that is if changes have been detected. This step is performed at last because redrawing is most efficient if it is done as a whole.

For efficiency reasons, changes are not updated immediately. It is quite costly to create a new node in GraphEd's data structure; for example, a bitmap has to be created, and it has to be repainted. But, many graph algorithms insert dummy nodes and edges which exist only temporarily and do not need to be displayed on screen. On the other hand, the cost of looking for changes is neglectable and is avoided completely if appropriate hints are supplied.

Since the interface between Sgraph and GraphEd consists of only a few, powerful functions, it is easy to extract algorithms and port them to another system. Only the user interface dependent part may need a larger rewrite, depending on the target system.

Application modules may also access several other facilities:

- They may change the graphical appearance of nodes and edges, via the core system's graph structure. The graphics system is fast enough to permit even algorithm animation and moving nodes and edges around in real time.
- Algorithm modules can use GraphEd's command dispatcher. The command dispatcher is the central part of the user interface and is responsible for performing all actions. For example, the dispatcher can be used create working area windows, change defaults or to load files. GraphEd's file selector can be used to choose a file.
- They may run external programs. The application interface provides a standard module for such tasks. GraphEd can communicate with other programs in two ways. Either the other program can read or write GraphEd's file format, or GraphEd is extended with a module which reads or writes a foreign format.
  This enables us to run algorithms on a fast machine over the network, or to use algorithms that have been implemented with separate systems. GraphEd may also be used as a graphical front end to other systems.
- They may add new windows to the user interface. GraphEd submodules may be completely self contained applications with their own user interface that just use GraphEd as a graph editor subsystem.

Besides installation in the menu, algorithm modules may also modify the tracking of mouse operations. For example, one can prevent node movements that destroy a planar drawing, or install application specific handlers for double clicks.

5.1 The Global Structure

An application module is a set of C modules which are compiled and linked with the core system of GraphEd. We choose this approach over a set of separate communicating programs like in SPREMB [5] since that needs extensive inter-process communication. This can degrade performance significantly, especially in an interactive system which depends on fast response time to the user's actions. The application interface would also be much more complicated, and probably less powerful. Moreover, our approach allows us to use external programs too, and there is also an interface available. We are also looking forward to provide network interfaces to other graph algorithm systems.

We have highly automated all maintenance tasks like building and installing the system or packing a distribution. To provide a convenient integration, we define an interface schema where startup code and external definitions are put in special files, which can be identified by name and automatically managed.

5.2 The File Format

GraphEd's file format is based on adjacency lists with embedded attributes. All GraphEd specific data is enclosed in **{$** and **$}**. Our approach has several major advantages. Programs which are only interested in the graph structure may extract it by ignoring everything between **{$** and **$}**. Programs which write input graphs for GraphEd do not need to supply the full set of attributes. Also, other programs can use the same technique to add their own attributes, since GraphEd's parser treats any text other in curly brackets as comments.

As a consequence, it is very easy to write an interface for reading and writing GraphEd files. It is also a simple task to add new attributes and guarantee backwards compatibility.

6 Layout in the Small: How to Draw Nodes and Edges

Anyone who uses a standard graphics program to draw graphs immediately runs into problems. If the size of a node changes, its label, all its adjacent edges and their labels need to be repositioned by hand. If the node is moved, the same adjustments have to be done. This comes from the fact that the graph structure imposes a lot of restrictions on the drawing, which are not handled by graphics programs.

Specifically for GraphEd, this can be divided into two categories:

- If many attributes of a node or edge are changed, they will likely affect the same internal graphics data structures. For example, if both the size and the position of a node change, then the position of the label must be recomputed each time.
  One part of GraphEd's solution is to delay the computation of the graphics until all changes are completed. For example, if a new graph is loaded, there will be a lot of changes, so the graphic data structures are computed as a whole at the end of the loading process.

- Similar problems arise if attributes of different objects are changed. For example, imagine that both end nodes of a straight line edge are moved independently. Then the position of the edge label must be recomputed for each endpoint. This will happen quite often with a graph layout algorithm or in algorithm animation.

If we take each attribute of each object separately, we would do a lot of unnecessary computations. In the following, we present a general approach to solve this problem.

We introduce a distinction between attributes and graphics intrinsics. The attributes control how a node or edge is drawn, whereas the intrinsics are the data structures that are needed to do the drawing. If an attribute gets changed, all its associated intrinsics must be recomputed. The relation between attributes and graphics is represented by the function

$$AttrInt : Attribute \mapsto \mathcal{P}(Intrinsics)$$

where *Attributes* is the set of node and edge attributes. If $AttrInt(a) = \{i_1, \ldots, i_n\}$, then the intrinsics $i_1, \ldots, i_n$ must be recomputed if the attribute a has changed. For example, $AttrInt(NodeSize) = \{NodeWidth, NodeHeight\}$.

There is more than that. If one intrinsic has changed, this can mean that other intrinsics have to be recomputed, too. For example, if the width of a node changes, its bounding box, its label position and *after* that the position of its adjacent edges and their labels have to be recomputed. The intrinsics dependency relation $<_I$ describes these facts:

$$<_I : Intrinsics \times Intrinsics \mapsto Boolean$$

$x <_I y$ means "if x has been recomputed, then y must be recomputed *after* x has been computed". $<_I$ must be acyclic. Figure 2 shows all intrinsics which are dependent on NodeWidth and NodeHeight.

6.1 Computing the Intrinsics

For simplicity, we assume that each intrinsic can be computed in one pass. This can be resolved by splitting an intrinsic i into $i_1, \ldots, i_k$ with $i_1 <_I i_2 <_I \ldots <_I i_k$. We do not consider more complex constructs because they would not be efficient anymore.

The intrinsics dependency graph $ID(G)$ of a graph G is a directed graph which is then constructed as follows:

```
**foreach** node or edge $x$ in G **do**
  **foreach** $i \in intrinsics(x)$ **do**
    **insert** node $n$ in $ID(G)$
    $intr(n) := i$; $graph(n) := x$
  **od**
**od**
```

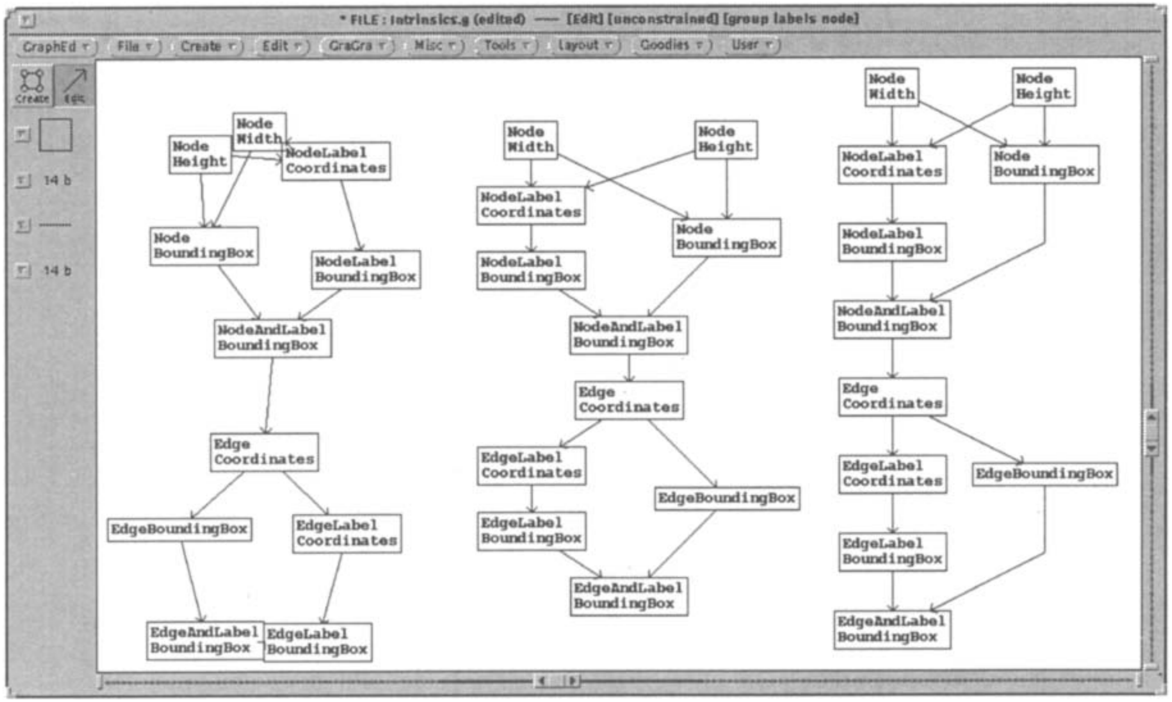


Fig. 2. Three versions of a graphical description of the intrinsics dependency relation $<_I$ starting at “NodeWidth” and “NodeNeight”. The left diagram is drawn with one of GraphEd’s force directed drawing algorithms, the right one with one for directed acyclic graphs, and the middle one is a drawing by hand.

```
foreach $i_1, i_2 \in nodes(ID(G))$ do
  if $(intr(i_1) <_I intr(i_2)) \wedge neighbour(graph(i_1), graph(i_2))$ then
    insert edge $(i_1, i_2)$ in ID(G)
  fi
od
```

where *intrinsics*(x) is the set of intrinsics of the node or edge x, and *neighbour*(x, y) is defined as follows:

$$\begin{aligned} neighbour(x,y) := {} & (x = y) \vee \\ & (x \in nodes(G) \wedge y \in adjacentedges(x)) \vee \\ & (y \in edges(G) \wedge x \in endnodes(x)) \end{aligned}$$

Let $\{(o_1, a_1), \ldots, (o_n, a_n)\}$ be the of objects and attributes that have changed, where attribute a_i is at object o_i. The set of nodes I' in ID(G) that correspondent to the intrinsics which need to be recomputed is now calculated as follows:

$$I := \{i \in nodes(ID(G)) \mid \exists j, 1 \le j \le n : graph(i) = o_j \wedge intr(i) \in AttrInt(a_j)\}$$
$$I' := \{i' \in nodex(ID(G)) \mid \exists i \in I : \exists\, path\, i \to^* i'\, in\, ID(G)\}$$

The actual recomputation is done with a topological sort on I':

```
$computed := \emptyset$
while $computed \neq I'$ do
  $ready := \{i \in I' \mid \forall (x,i) \in edges(ID(G)) : x \in computed\}$
  compute the intrinsics in $ready$
  $computed := computed \cup ready$
od
```

The termination is guaranteed by the fact that $<_I$ is acyclic. For efficiency reasons, the actual implementation in GraphEd does not use the dependency graph, but a hand coded version which is optimized for the data structures that are used in GraphEd.

The fact that the underlying structure is a graph is only used in *neighbour*. This means that our approach can easily be extended to handle arbitrary structures by redefining *neighbour*. For example, hypergraphs or graphs with half edges (that is edges with only one endpoint), or hierarchical graphs as described in [10] can easily be modelled. It is also not limited to the description of graphics.

7 Layout in the Large: Graph Drawing

At some point of the implementation of most graph algorithms, there comes the need to actually draw a graph. However, coordinates are available only if the graph came with the layout coordinates, and has not been changed too much. Otherwise, an automatic layout algorithm is needed. This might be the case if a new graph is constructed, or an intermediate graph structure needs to be visualized for debugging. Also, layout algorithms are an important tool to support users while editing graphs.

An overview of the current state of graph drawing algorithms is presented in [4]. One major problem is that most of the criteria upon which the current algorithms are based are NP complete [2], so heuristics are needed. Another problem is that the exact relevance of the criteria is yet unknown. [19] and [12] survey that topic and show some results on ranking layout criteria.

GraphEd comes with a set of parametrized layout algorithms which cover a wide range of the algorithms that are described in the literature. [12] provides an in-depth discussion of several of the algorithms which are implemented in GraphEd:

- Algorithms for general graphs:
  - Several algorithms for force directed drawing
  - Directed acyclic graph drawing
- Algorithms for planar graphs
  - Planar grid drawing with rectangular edges and bends minimization
  - Planar grid drawing with straight line edges
  - Planar grid drawing
  - Planar convex faces drawing with straight line edges
  - Several algorithms for visibility representations
  - ST upward drawing algorithms
- Algorithms for graphs with a special structure
  - Tree drawings
  - Several graph grammar based layout algorithms.

8 Conclusion

GraphEd has grown to over 240000 lines of C code, 80 % of which are application modules contributed by several people. Sgraph has proved to be a flexible data structure that can be adapted to various applications. GraphEd has been used at various sites as a tool to develop and implement a large range of algorithms. The actual applications include (see [10] for an overview):

- Graph editor.
- Development of graph algorithms, including algorithm animations.
- Development and evaluation of graph layout algorithms [11].
- Combinatorial algorithms.
- Graph grammar algorithms, including a parser and a layout system.
- Application of graph grammars.
- Data base design.
- Graphical front-ends.

Planned work includes advanced graphics and user interface, extensions to the application interface and new layout technologies that are based on the experiences we could gather with the existing ones.

References

1. Bordier, J.: Cabri 3.0 Reference Guide. Laboratoire de Structures Discretes et de Didactique, Universite Joseph Fourier & C.N.R.S., Grenoble (1990)
2. Brandenburg, F.J.: Nice drawings of graphs are computationally hard. Technical report, Universität Passau (1988)
3. Brandenburg, F.J., Himsolt, M., Röeder, F.J., Skodinis, K.: Designing dataflow graphs by graph grammars. In Colloquium on Graph Transformation and its Application in Computer Science, Universitat de les Illes Baleares, Departament de Ciences Matematiques i Informatica (1994)
4. di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Algorithms for drawing graphs: An annotated bibliography. Technical report, Dept. of Computer Science, Brown Univ. (1993)
5. Eades, P., Fogg, P. and Kelly, D.: Spremb: a system for developing graph algorithms. Congressus Numerantium, **66** (1988) 123–140
6. Ehrig, H., Nagl, M. Rosenfeld, A., Rozenberg, G. (editors): Graph-Grammars and Their Application to Computer Science, LNCS **291** (1987)
7. Fröhlich, M., Werner, M.: The interactive graph visualization system davinci v1.2. Technical report, Universität Bremen (1993)
8. GraphEd and related documents are available by anonymous ftp from the site `ftp.uni-passau.de`, `/pub/local/graphed`.
9. Himsolt, M.: GraphEd: An interactive tool for developing graph grammars. Graph Grammars and Their Application To Computer Science, LNCS **532** (1990) 61–65
10. Himsolt, M.: Konzeption und Implementierung von Grapheneditoren. Dissertation, Universität Passau, 1993.
11. Himsolt, M.: A view to graph drawing algorithms through GraphEd. Proceedings of the ALCOM International Workshop on Graph Drawing and Topological Graph Algorithms (1993)
12. Himsolt, M.: Comparing and evaluating layout algorithms with GraphEd. Manuscript, available via ftp from `ftp.uni-passau.de` (1994)
13. Knuth, D.E.: The Stanford GraphBase: A Platform for Combinatorial Algorithms. Stanford University (1993)
14. Design Users Guide Meta Software Corporation, Cambridge, MA (1987)
15. Näher, S.: LEDA Manual. Max-Planck-Institut für Informatik, Saarbrücken (1993)
16. Newberry Paulisch, F.: The Design of an Extendible Graph Editor, LNCS **704** (1993)
17. Rowe, L.A., Davis, M., Messinger, E., Meyer, C., Spirakis, C., and Tuan, A.: A browser for directed graphs. Software Practice and Experience, **17(1)** (1987) 61–76
18. Schürr, A.: Progres: A vhl-language based on graph grammars. Graph Grammars and Their Application to Computer Science, LNCS **532** (1990) 641–659
19. Tamassia, R., di Battista, G., Batini, C.: Automatic graph drawing and readability of diagrams. IEEE Transactions on Systems, Man and Cybernetics **18(1)** (1988) 61–79