GD-Workbench: A System for Prototyping and Testing Graph Drawing Algorithms*

Luciano Buti[1], Giuseppe Di Battista[2], Giuseppe Liotta[3], Emanuele Tassinari[1], Francesco Vargiu[1], and Luca Vismara[4]

[1] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
{buti,tassinar,vargiu}@dis.uniroma1.it
[2] Dipartimento di Discipline Scientifiche, Sezione Informatica
Terza Università di Roma
Via Segre 2, 00146 Roma, Italy
dibattista@iasi.rm.cnr.it
[3] Department of Computer Science, Brown University
115 Waterman Street, Providence, Rhode Island 02912–1910
gl@cs.brown.edu
[4] Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche
Viale Manzoni 30, 00185 Roma, Italy
vismara@iasi.rm.cnr.it

Abstract. We present a tool for quick prototyping and testing graph drawing algorithms. The user interacts with the system through a diagrammatic interface. Algorithms are visually displayed as directed paths in a graph. The user can specify an algorithm by suitably combining the edges of a path. The implementation exploits the powerful functionalities of Diagram Server and has been experimented both as a research support tool and as a back-end of an industrial application.

1 Overview

We present GD-Workbench (GDW), a system for quick prototyping and testing graph drawing algorithms. The user interacts with GDW through a diagrammatic interface. Graph drawing algorithms are visually displayed as directed paths in a graph. The potential users of GDW are:

- Graph drawing *researchers* that aim at *experimenting* existing algorithms against real-life or randomly generated graphs; an experimental work performed with an early version of GDW is described in [2].
- Graph drawing *researchers* that aim at *implementing* new algorithms and at quick understanding how such algorithms can exploit existing algorithms as subroutines.

* Work supported in part by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of the Consiglio Nazionale delle Ricerche, by ESPRIT Basic Research Action No. 7141 (ALCOM II), by the National Science Foundation under grant CCR-9423847, and by N.A.T.O.- C.N.R. Advanced Fellowships Programme.

- Graduate course *teachers* that aim at easily demonstrating in their classes the behavior of graph drawing algorithms.
- *Professionals*, already skilled with graph drawing, that want to select the best algorithm for a given application; as an example, GDW has been used in cooperation with the *Italian Authority for Computer Engineering in the Public Administration*: about 2,500 Entity-Relationship diagrams have been automatically drawn and the whole work (algorithm-selection, graphic features setting, fine-tuning, large-scale drawing, and printing) has been performed with GDW.

GDW has the following main functionalities:

- *Path-management.* It allows to construct a new algorithm by composing steps of existing algorithms. The new *algorithm* is visually represented as a *path in a graph.* It also allows to visualize several algorithms at the same time, to discard previously visualized algorithms, and to show info on algorithms.
- *Schema-management.* It allows to load graphs (we will use the terms schema and graph as synonyms) and to create them either with a graph editor or with random graph generators.
- *Test-management.* It allows to draw the current graphs with the currently visualized algorithms and to generate reports on the aesthetic features of the drawings.

Although several interesting and powerful tools have been recently devised in the graph drawing field (to give only a few examples we mention [5, 4, 7, 6, 3]), we believe that GDW has several innovative characteristics. In particular, the user-interaction paradigm of GDW has flexibility and friendliness features that, to our knowledge, have no counterparts in existing tools. The flexibility and friendliness of GDW are both in providing an easy interaction with the algorithms and in showing diagrams. Existing tools usually focus on just one of these two aspects.

Rather than presenting all the details of the architecture and of the implementation, we prefer to start the description of the system with an introductory example (Section 2). However, the main architectural issues are outlined in Section 3. Section 4 describes further examples of usage of GDW. Future research directions are sketched in Section 5.

The paper is supplied with several figures, most of them snapshots of the screen.

2 An Introductory Example

We show how a user can simply construct his/her own graph drawing algorithm with GDW, by combining pieces of existing algorithms.

2.1 The Taxonomy

GDW presents the algorithms to the user through a *taxonomy* of classes of graphs. The most general class of graphs of the taxonomy is *Multigraph*; a multigraph is a graph that has both directed and non-directed edges. All the other

classes of the taxonomy are subclasses of *Multigraph*. Each class is provided with a set of *methods* that map an object of a class into an object of another class. A method is a layout functional step, taken from an existing algorithm. A drawing algorithm A is a sequence of methods that is visually represented on the taxonomy as a path (*algorithmic path*); the edges of the algorithmic path describing A are the methods that compose A and the vertices are the classes of the taxonomy the methods are associated to.

The taxonomy is a very general structure to classify graph drawing algorithms and has been already exploited for the internal structure of the algorithms database of Diagram Server [1, 3].

2.2 Constructing a New Algorithm

Suppose the user wants to draw graphs with a polygonal graphic standard (i.e. all edges are polygonal lines) and to this aim wants to construct a "new" algorithmic path. He/she opens a window displaying the taxonomy and executes the following steps:

- All the classes of the taxonomy are displayed on the screen. The dashed edges of the taxonomy show containment between classes. Class *Multigraph* is white colored. White classes (in this case only *Multigraph*) are the already selected classes for the algorithmic path (Fig. 1).
  The classes *Connected*, *Planar*, and *Digraph* are red colored. Red classes are the ones that can be reached by applying an available method of the last selected class of the currently constructed algorithmic path (in this case class *Multigraph*).
- The user clicks on one of these three classes, say *Planar*. The system displays the set of available methods that transform an element of *Multigraph* into an element of *Planar*. The user can now select one method of the set. In this case the set consists of just one method, namely `MakePlanar` (Fig. 2).
- Now class *Multigraph* is white, class *Planar* is colored white and red, and the classes *Connected*, *Digraph*, *ConnectedPlanar*, and *FourPlanar* are red. *Planar* is white and red because (i) it is already selected for the algorithmic path and (ii) there is method `MakePlanar` (inherited from the class *Multigraph*) that transforms an element of *Planar* into an element of the class itself. The user can now click on any reachable class, i.e. either a red or a white and red class, say *ConnectedPlanar* (Fig. 3).
  The system displays the set of methods that transform an element of *Planar* into an element of *ConnectedPlanar*. The user can now select one method of the set. In this case the set consists of two methods, namely `MakeConnected` and `IsConnected`. In our example, the user selects `MakeConnected`.
- The user, by performing other similar operations, constructs an algorithmic path whose final class is *Polygonal*. The complete path is now on the screen and consists of classes *Multigraph*, *Planar*, *ConnectedPlanar*, *BiconnectedPlanar*, *PlanarSTDigraph*, *ReducedPlanar*, *Straightline*, and *Polygonal*, plus the methods connecting them (Fig. 4).

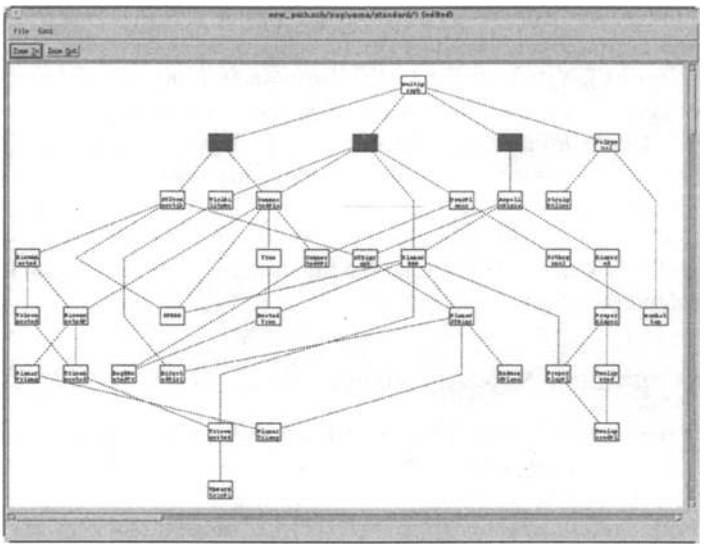

Fig. 1.

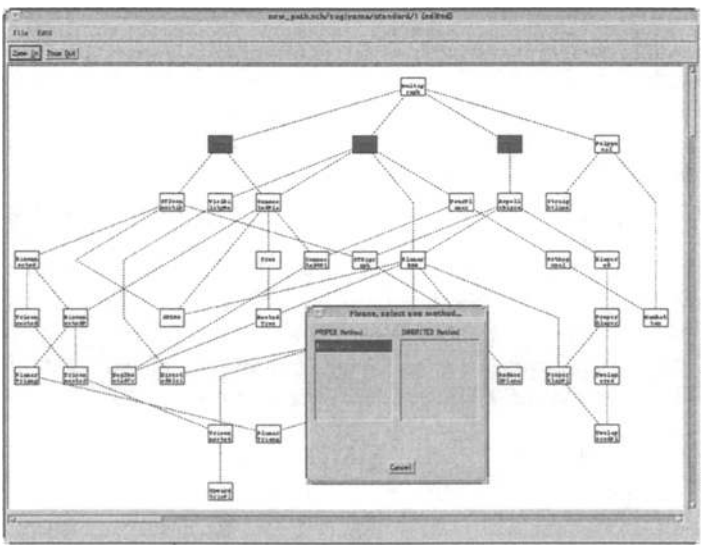

Fig. 2.

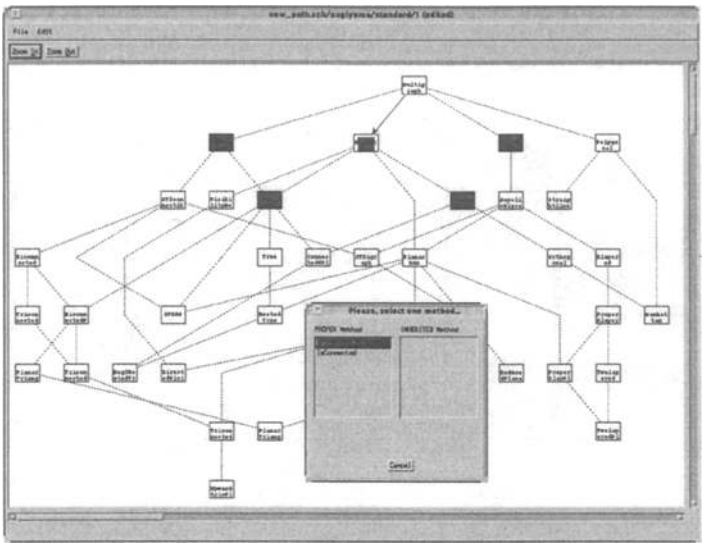

Fig. 3.

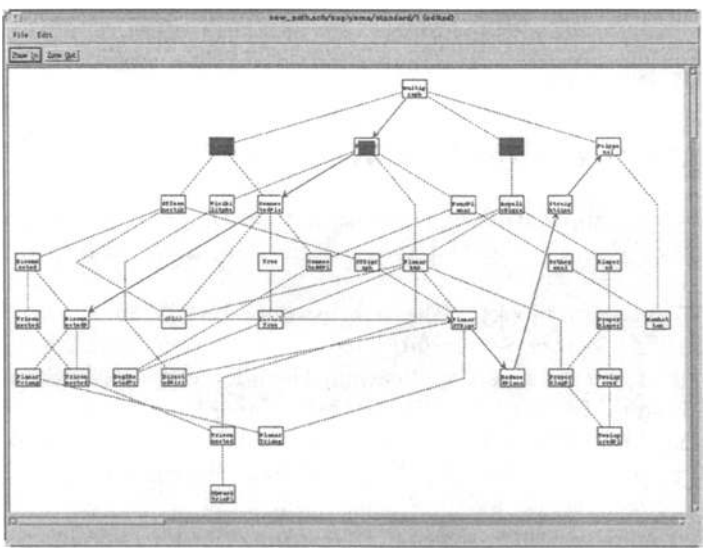

Fig. 4.

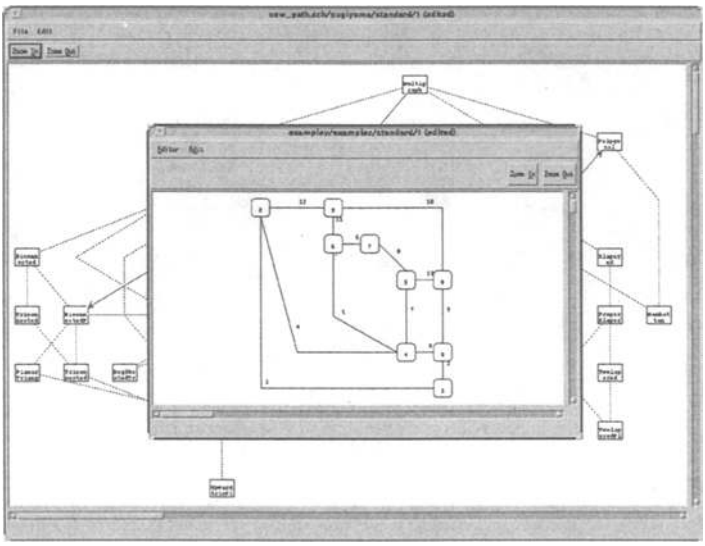

Fig. 5.

– The user can now execute the algorithmic path to obtain a drawing of an input graph (Fig. 5). Of course, if he/she is not satisfied by the resulting drawing, with similar operations the algorithmic path can be modified, choosing different classes and/or different methods. The algorithmic path can also be stored to be reused.

3 The Architecture of GDW

GDW is a client application of Diagram Server. Diagram Server provides the capabilities for the drawing and the visualization of the graphs managed by GDW.

In Fig. 6 the main blocks of the architecture of GDW are shown.

The *GDW / Diagram Server Interface* coordinates the exchange of data between the client and the server application. The interaction is based on a message passing technique. For example: (i) GDW can force Diagram Server to wait for a user action; (ii) Diagram Server can notify GDW that the user has clicked on a menu item; (iii) GDW can force Diagram Server to enter the status in which vertices and edges can be added or deleted; etc.

The *Schema Manager* provides the functionalities for loading, creating (automatically or manually), and discarding schemas. Once a schema is loaded, it is active and it can be used during the testing of the algorithmic paths.

A schema can be randomly generated by using the *Random Graph Generator* of GDW. It is possible to generate different types of graphs, including connected graphs, biconnected planar graphs, and trees. The component uses two different strategies for the generation of the graphs: (i) they are generated from scratch,

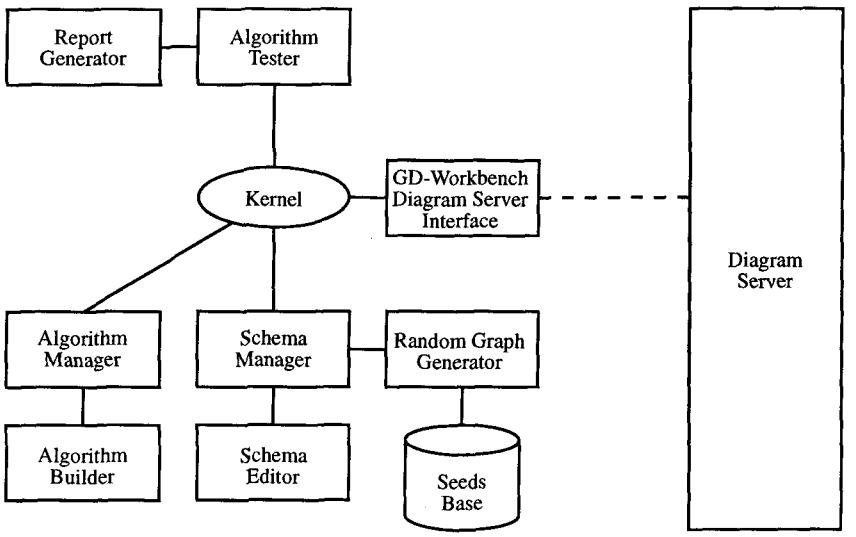

Fig. 6. The Architecture of GDW.

by using, first, randomly insertion of edges and, second, local adjustments that force them to belong to the chosen class; (ii) they are generated starting from a core set of existing real-life based graphs (stored in the *Seeds Base*), by means of a list of operations that preserve the similarity with the starting graph.

Alternatively, a schema can be constructed by using the *Schema Editor*, a powerful interactive editor that allows to add, delete, move, reshape vertices and edges.

The *Algorithm Manager* manages the algorithmic paths. They can be loaded, created, visualized, and discarded. As for the schemas, once an algorithmic path is loaded, it becomes active.

The *Algorithm Builder* is the component devoted to the creation of new algorithmic paths. Algorithmic paths can be created also starting from existing ones. The Algorithm Builder, through the GDW / Diagram Server Interface, asks Diagram Server to display the taxonomy on a window. The actions performed by the user on the taxonomy are captured by Diagram Server and notified to the tool, that executes the proper operations (e.g. opening of a dialog window, sending a message for highlighting a class of the taxonomy, sending a message for the insertion of an edge).

The *Algorithm Tester* and the *Report Generator* are the components for the testing and the evaluation of the active algorithmic paths on the active schemas. Partial and full reports on the tests can be generated as well as diagrams with disparate graphic features.

4 Further Examples

In this section we exploit the functionalities of GDW by means of a set of examples.

The first example shows the facilities to manage algorithmic paths.

- AlgorithmicPaths menu allows to activate, discard, create, highlight, get info on algorithmic paths.
- ActivatePath item of AlgorithmicPaths menu displays a dialog window with a set of algorithmic paths that have been stored in previous working sessions. Once an algorithmic path is selected, it is activated, it is shown in the taxonomy, and it can be executed on a given set of graphs.
- Displaying several active algorithmic paths. Different colors identify different algorithmic paths. White classes and edges describe subpaths that are shared by two or more algorithmic paths. For example, the yellow path is the algorithm `bend2` (Fig. 7).
- Info about the active algorithmic paths. A dialog window is shown with the correspondence between colors and algorithmic paths. By selecting one color, the list of classes and methods of the corresponding algorithmic path is displayed in a text window (Fig. 8).
- Highlighting an algorithmic path. When several algorithms are shown at the same time, the screen may become difficult to read. Each active algorithmic path can be highlighted by selecting it in a dialog window.
- By clicking on a class (say *PlanarTriangulated*) the system displays the available methods for that class and for each method suitable bibliographic references (Fig. 9).

The following example illustrates the capabilities of GDW in managing graphs.

- Schemas menu allows to activate, discard, and get info either on single schemas or on directories of schemas.
- ActivateSchema item of Schemas menu displays a dialog window with the stored schemas is shown. Once a schema is selected it can be drawn using the active algorithmic paths.
- InfoSchemas. The identifiers of the active schemas are shown on a dialog window.
- Random graph generation. GDW is provided with a random graph generator. The user, by means of a dialog window, can select the class of graphs to be generated, their number and size. In this case two biconnected graphs with ten vertices are generated (Fig. 10).
- Alternatively, the user may construct graphs by using a powerful interactive graph editor.

The functionalities of GDW in testing algorithms are shown by the following example.

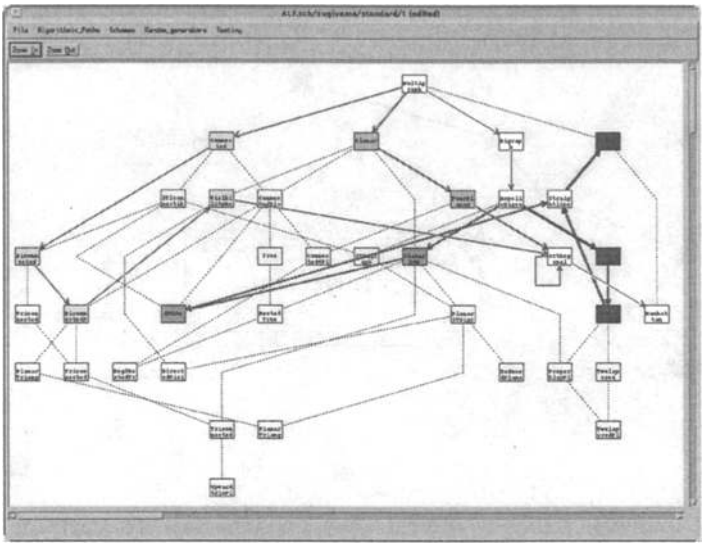

Fig. 7.

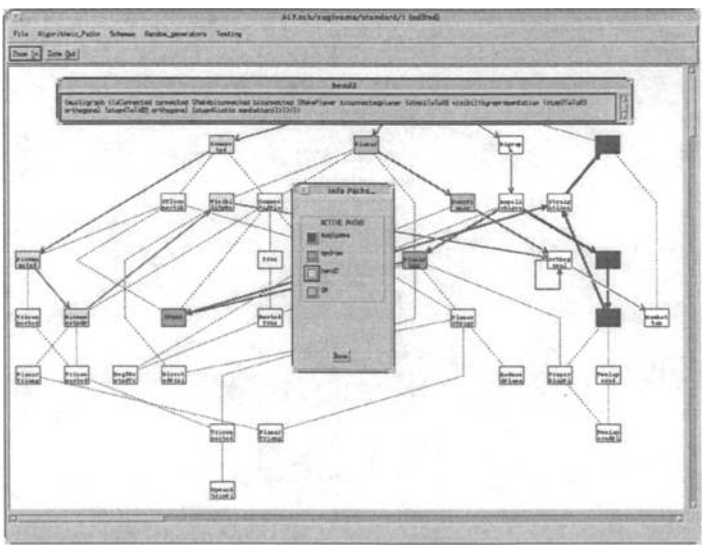

Fig. 8.

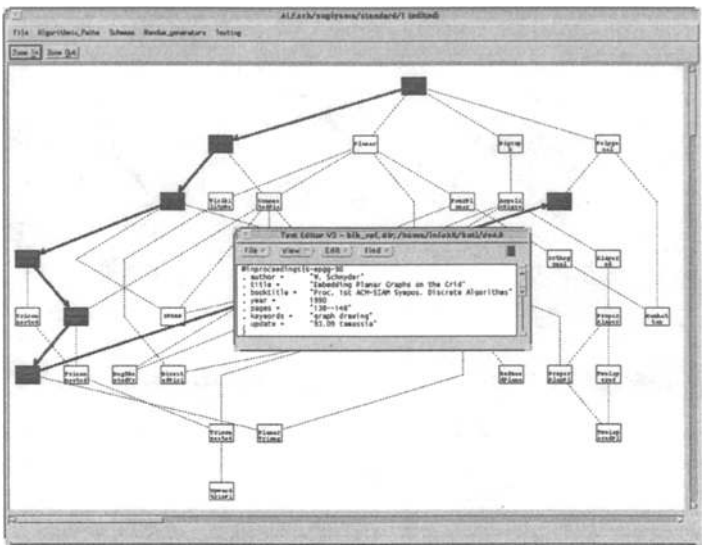

Fig. 9.

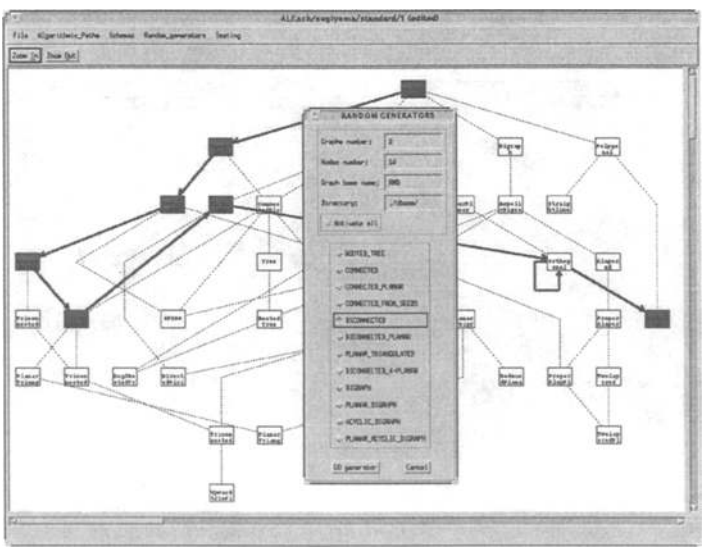

Fig. 10.

- Testing menu allows to apply the active algorithmic paths on the active schemas (randomly generated or manually constructed) and to setup several output options.
- Output Options item of Testing menu displays a dialog window in which the user can select the diagrams to be displayed on the screen and choose to generate the reports.
- The diagrams obtained by applying the active algorithmic path (`bend2`) to two randomly generated graphs are shown in Fig. 11.
- In the left window of Fig. 12 the reports on the previous application are displayed, while in the right window the average results of the reports are displayed.

5 Future Work

We will improve and expand our tool in the following directions.

- We plan to interconnect GDW with an object-oriented software development platform, in order to support the whole development cycle of a graph drawing algorithm.
- In order to better show experimental reports, we aim at integrating into GDW a system for visualizing graphics.
- We aim at extending our experiments by further interacting with the Italian Public Administration, which is a promising source of case studies.

References

1. P. Bertolazzi, G. Di Battista, and G. Liotta. Parametric graph drawing. *IEEE Trans. Softw. Eng.*, 21(8):662–673, 1995.
2. G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of three graph drawing algorithms. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995.
3. G. Di Battista, G. Liotta, and F. Vargiu. Diagram Server. *J. Visual Languages and Computing* (special issue on Graph Visualization, I. F. Cruz and P. Eades, editors), 6(3), 1995.
4. C. Ding and P. Mateti. A framework for the automated drawing of data structure diagrams. *IEEE Trans. Softw. Eng.*, SE-16(5):543–557, 1990.
5. P. Eades, I. Fogg, and D. Kelly. SPREMB: a system for developing graph algorithms. *Congressus Numerantium*, 66:123–140, 1988.
6. M. Himsolt. GraphEd: A graphical platform for the implementation of graph algorithms. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pages 182–193. Springer-Verlag, 1995.
7. F. N. Paulish and W. F. Tichy. EDGE: An extendible graph editor. *Softw. – Pract. Exp.*, 20(S1):63–88, 1990.

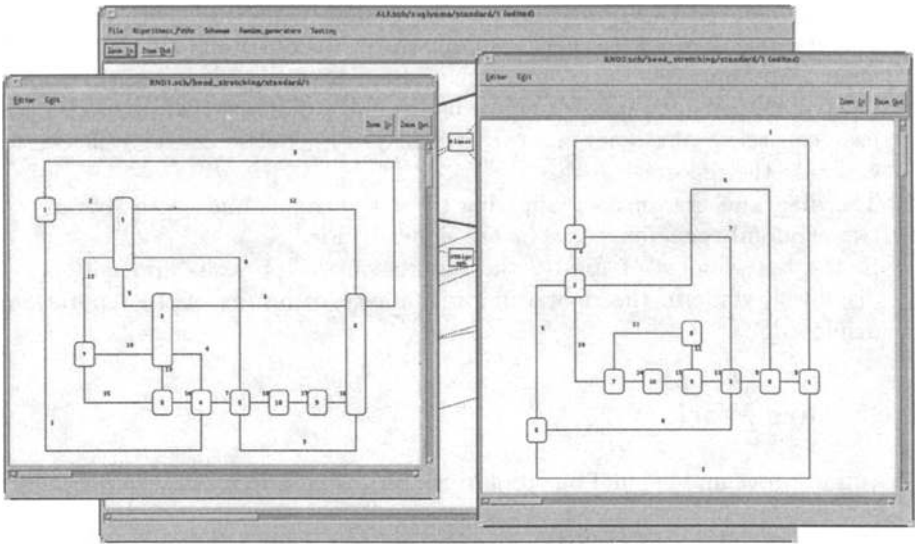

Fig. 11.

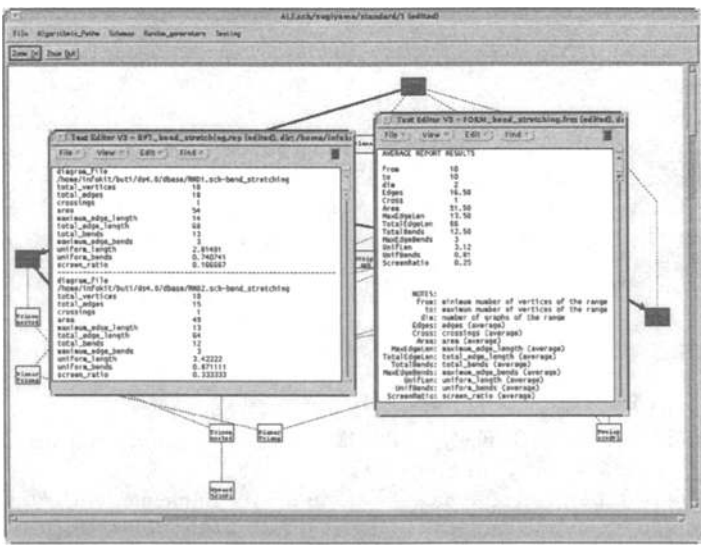

Fig. 12.