

NAVAL POSTGRADUATE SCHOOL MONTEREY, CALIFORNIA



19980414 101

THESIS

**AUTOMATIC LAYOUT TECHNIQUES FOR THE
GRAPHICAL EDITOR IN THE COMPUTER AIDED
PROTOTYPING SYSTEM (CAPS)**

by

William J. Ray

September, 1997

Thesis Advisor:

Luqi

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 1997		3. REPORT TYPE AND DATES COVERED Engineer's Thesis
4. TITLE AND SUBTITLE. AUTOMATIC LAYOUT TECHNIQUES FOR THE GRAPHICAL EDITOR IN THE COMPUTER AIDED PROTOTYPING SYSTEM (CAPS).			5. FUNDING NUMBERS	
6. AUTHOR(S) William J. Ray				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) The Computer Aided Prototyping System (CAPS) is a systems engineering tool intended to make the iterative process of software development more efficient. The simplest way to input and modify a CAPS design is through the graphical editor. When a design is modified over and over, the resultant graphical representation can become difficult to comprehend. Trying to change the graphical representation by hand can be very tedious. By adding automatic layout techniques to the graphical editor, this task is made easier for the user of the system. Automatic layout techniques for general graphs that maximize all of the aesthetic characteristics of a graph are not possible. One characteristic may conflict with another. By giving the user multiple layout algorithms that emphasis different characteristics over others, the user may choose between different layouts for the graphical representation. Since CAPS was in the middle of a restructure and no graphical editor was available, automatic layout techniques were investigated using other graphical editors. Graphs with characteristics similar to a CAPS graph were input into the graphical editors and then the layout algorithms applied. The results of this assessment proved that the addition of automatic layout techniques to CAPS would improve performance. The library of layout algorithms will be incorporated into the new graphical editor in CAPS.				
14. SUBJECT TERMS Topology, Graph, Layout, CAPS			15. NUMBER OF PAGES 324	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

Approved for public release; distribution is unlimited.

**AUTOMATIC LAYOUT TECHNIQUES FOR THE GRAPHICAL EDITOR
IN THE COMPUTER AIDED PROTOTYPING SYSTEM (CAPS)**

William J. Ray
B.S., Purdue University, 1985

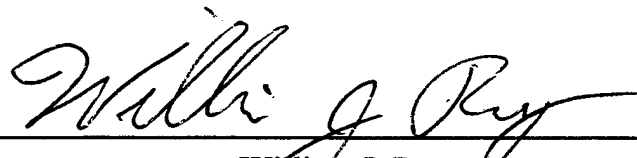
Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
September 1997**

Author:

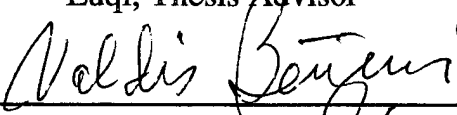


William J. Ray

Approved by:



Luqi, Thesis Advisor



V. Berzins, Second Reader



T. Lewis, Chairman
Department of Computer Science

ABSTRACT

The Computer Aided Prototyping System (CAPS) is a systems engineering tool intended to make the iterative process of software development more efficient. The simplest way to input and modify a CAPS design is through the graphical editor. When a design is modified over and over, the resultant graphical representation can become difficult to comprehend. Trying to change the graphical representation by hand can be very tedious.

By adding automatic layout techniques to the graphical editor, this task is made easier for the user of the system. Automatic layout techniques for general graphs that maximize all of the aesthetic characteristics of a graph are not possible. One characteristic may conflict with another. By giving the user multiple layout algorithms that emphasize different characteristics over others, the user may choose between different layouts for the graphical representation.

Since CAPS was in the middle of a restructure and no graphical editor was available, automatic layout techniques were investigated using other graphical editors. Graphs with characteristics similar to a CAPS graph were input into the graphical editors and then the layout algorithms applied. The results of this assessment proved that the addition of automatic layout techniques to CAPS would improve performance. The library of layout algorithms will be incorporated into the new graphical editor in CAPS.

TABLE OF CONTENTS

I. INTRODUCTION.....	1
A. GENERAL.....	1
B. PROBLEM STATEMENT	2
C. SCOPE	2
II. BACKGROUND KNOWLEDGE	3
A. GENERAL.....	3
B. GRAPH CHARACTERISTICS	3
C. AUTOMATIC LAYOUT TECHNIQUES	5
1. Minimize crossing.....	6
2. Minimize area.....	6
3. Minimize bends (in orthogonal drawings).....	6
4. Minimize slopes (in polyline drawings).....	6
5. Maximize smallest slope	6
6. Maximize display of symmetries	6
III. ANALYSIS OF AUTOMATED LAYOUT TECHNIQUES	9
A. GENERAL.....	9
B. GENERAL TECHNIQUES.....	9
C. ALGORITHM SEARCH.....	10
D. THE GRAPH DRAWING SERVER (GDS)	11
1. Giotto.....	11
2. Giotto with labels.	11
3. Bend-Stretch.	12
4. Pairs.....	12
5. Series Parallel Drawing.....	12
6. Sugiyama.....	12
7. Column.....	13
8. Ortho Upward.....	13
9. Ortho Non-Upward.....	13
10. Planarizer.....	13
E. GRAPHLET	14
1. Random Layout.....	14
2. Spring Embedder with Constraints	15
3. Iterative Constraint Spring Embedder	16
4. Spring Embedder (GEM).....	17
5. Spring Embedder (Kamada).....	18
6. General Graphs (Tunkelang).....	19
7. DAG.....	19
IV. ANALYSIS OF CAPS GRAPHICAL EDITOR.....	21

A. GENERAL	21
B. CAPS CHARACTERISTICS.....	21
V. FUTURE REQUIREMENTS OF AUTOMATED LAYOUT TECHNIQUES FOR DIRECTED GRAPHS	23
A. GENERAL	23
B. RESEARCH AREAS	23
1. <i>Performance Bounds for Planarization.</i>	23
2. <i>Simple Planarity Testing.</i>	24
3. <i>General Strategy for Straight-Line Drawings.</i>	24
4. <i>Dynamic Drawing Algorithms.</i>	24
5. <i>Complexity of Bend Minimization.</i>	24
6. <i>Angular Resolution of Planar Straight-Line Drawings.</i>	25
VI. NEW AUTOMATED LAYOUT TECHNIQUES FOR CAPS DESIGN AND IMPLEMENTATION	27
A. GENERAL.....	27
B. CURRENT PROBLEMS.....	27
C. DESIGN OF TEST SYSTEM	28
D. IMPLEMENTATION OF TEST SYSTEM	29
1. <i>PSDL Data Type.</i>	29
2. <i>Middleware.</i>	30
3. <i>GML File.</i>	30
4. <i>Graphlet system.</i>	30
E. DESIGN OF REAL SYSTEM.....	30
F. IMPLEMENTATION OF REAL SYSTEM.....	31
VII. CONCLUSION AND FUTURE RESEARCH	33
A. CONCLUSION.....	33
B. FUTURE RESEARCH.....	33
C. CAPS IMPROVEMENT	34
1. <i>Spline Drawing.</i>	34
2. <i>File format.</i>	34
APPENDIX A: LEDA LICENSE INFORMATION	35
A. LEDA LICENSE	35
B. LEDA INFORMATION	35
1. <i>LEDA-R-3.5.1</i>	35
2. <i>Differences between LEDA and LEDA-R</i>	36
3. <i>Who needs a license?</i>	36
C. LEDA SOURCE CODE	37
APPENDIX B: CODE	233
A. TEST CODE CREATED.....	233

B. NEW CODE CREATED	234
C. CAPS CODE MODIFIED	240
LIST OF REFERENCES.....	307
INITIAL DISTRIBUTION LIST	309

LIST OF SYMBOLS

O Order

I. INTRODUCTION

A. GENERAL

The chore of programming a computer to perform a desired task has become increasingly difficult over the past few decades. The tasks that computer programmers are asked to complete are much more complex and difficult than previously attempted. However, the tools that computer programmers have utilized to perform these tasks have not enabled these systems to be easily built and maintained.

The trend in the field of software engineering is towards automated tools that handle the more tedious components of the software engineering process. This allows the computer programmer to apply more resources to the creative process by trying different approaches without investing too many assets up front.

The Computer Aided Prototyping System (CAPS) is an integrated set of tools that allows a software engineer to design large software systems and test their design prior to implementation. The software engineer can start designing a system with CAPS using a top down approach. At each level of detail, the system can be tested to ensure that the system's performance is within the parameters of the desired end product.

To ease the process, CAPS has a graphical interface for the design process. Each operator can be laid out in the graphical display. These operators are connected to each other with directed lines that represent streams. Each operator can be subdivided into simpler operators until all the operators are atomic operators. Each operator can have performance constraints associated with it. CAPS can test that these constraints are not violated by the overall design. If a violation occurs, the design can be modified until the desired system specifications are reached.

B. PROBLEM STATEMENT

This research is a revision of the work on the graphical editor of CAPS. The current graphical editor has no automated layout capabilities. The program graph can become convoluted after many edits. Automated layout techniques will be investigated for use in the CAPS graphical editor that will minimize such attributes as crossing lines and spacing.

By adding automatic layout techniques to the graphical editor, designer/programmer productivity is gained by helping them get the job done faster. Further, the resulting program graphs will be easier to comprehend.

C. SCOPE

The scope of this thesis will deal with the capabilities of layout techniques for directed graphs while simultaneously analyzing the unique needs of the graphical editor in CAPS. Once both are completed, findings will be used to build an automated layout function that will meet the needs of the CAPS in the most user friendly means available.

This thesis contains three primary products. The first is a survey of current automated layout algorithms for graphs. The second is an evaluation of the properties of a common program graph in CAPS. The third is the implementation of automated layout algorithms for the CAPS graphical editor.

II. BACKGROUND KNOWLEDGE

A. GENERAL

The Computer Aided Prototyping System (CAPS) is a software-engineering tool designed to take the drudgery out of prototyping systems. A software developer can design a software system, test the design, and add timing requirements. Most projects become an iterative process of re-visiting the design, making the necessary changes to fix problems, and then testing the new design.

After a few iterations, the graphical display of the prototyped system can become very difficult to understand, since the graphical display requires the user to layout the design. Fixing the design layout by hand requires some time. The layout editor in CAPS is also rather slow and plodding for the user. By providing automatic layout techniques to the user, the chore of laying out the design in a more coherent manner is lessened.

B. GRAPH CHARACTERISTICS

All graphs have different characteristics. By testing a graph for these characteristics, a graph can be categorized into different graph categories. These graph categories include trees, planar undirected graphs, planar directed graphs, general undirected graphs, general directed graphs, etc.

Graphs can be trees if no cycles exist. They can be planar if the display of the graph is meant to be two-dimensional.

A general graph can be displayed with any-dimensional perspective. The edges connecting the graphs can also differ in category. In general, directed graphs have an edge with at least one arrow signifying direction. Undirected graphs have no arrows and there is no restriction on the direction. Figure 2.1 shows several different graphs with differing characteristics.

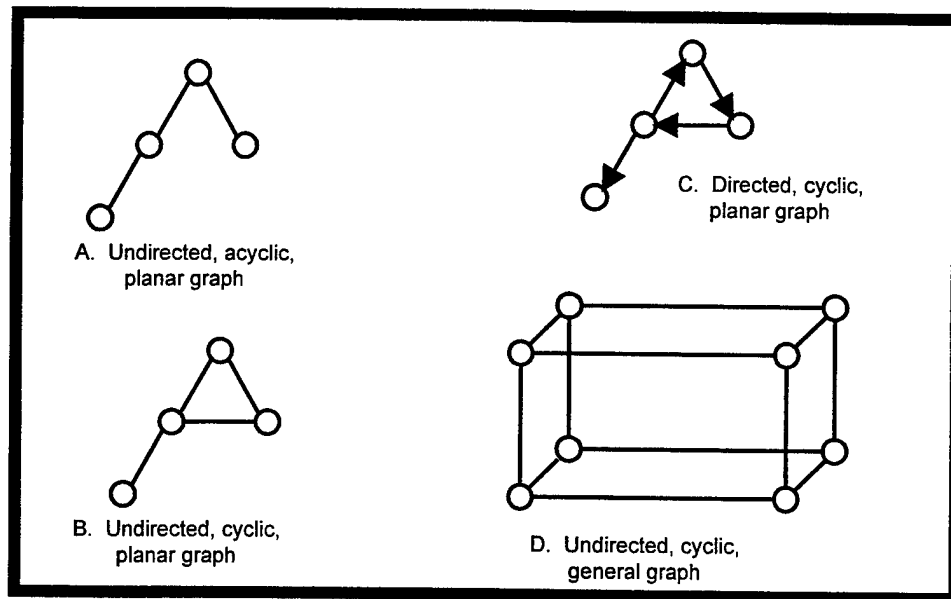


Figure 2.1: Different characteristics of graphs

Another characteristic important in laying a graph out is whether polyline drawings are permitted. Polyline drawings allow for bends in an edge. Straight-line drawings only allow for straight-lines to be drawn between vertices. Figure 2.2 demonstrates this problem.

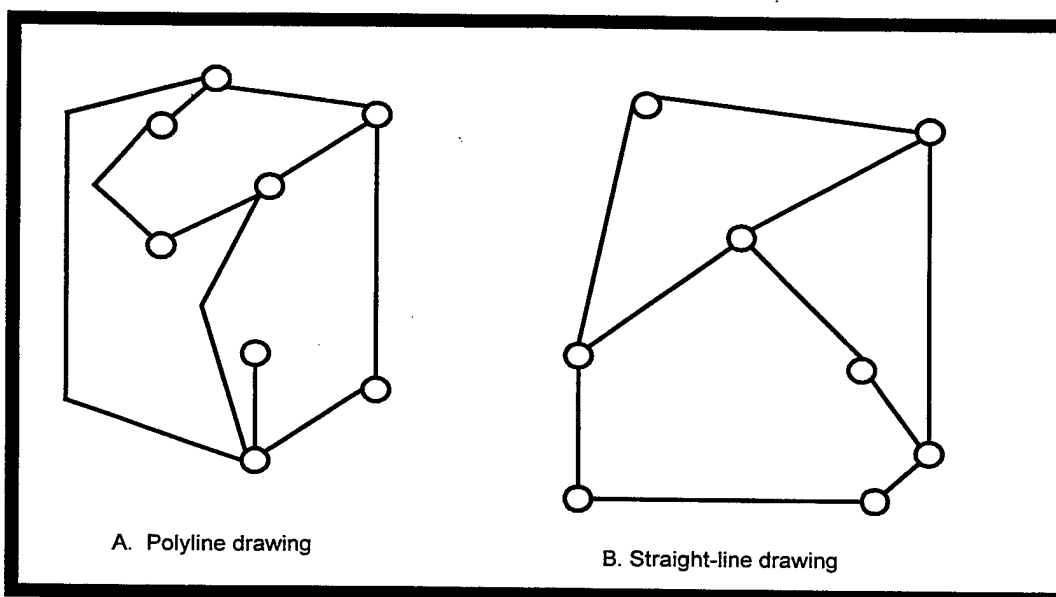


Figure 2.2: Polyline drawing vs Straight-line drawing

An orthogonal drawing maps each edge into a chain of horizontal and vertical segments. This produces a boxy looking graph. Figure 2.3 gives an example of an orthogonal graph.

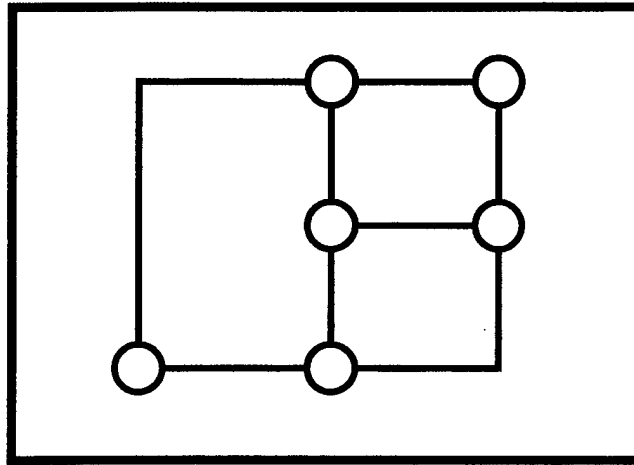


Figure 2.3: Orthogonal graph

C. AUTOMATIC LAYOUT TECHNIQUES

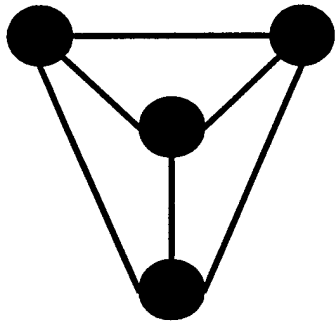
Automated layout techniques for a graph is an ongoing research area. Laying out a generic graph optimally is considered a NP Complete task. However, there exist algorithms that can approximate an optimal solution. By having the computer do most of the tedious work, a user optimize the layout with minor adjustments.

Graph layout techniques can be written to layout a generic graph or a subset of the set of all graphs. A tree graph would use an algorithm whose sole purpose is to optimize the layout of trees. This algorithm would not work for graphs that have cycles. Another class of algorithms would be used for graphs with cycles.

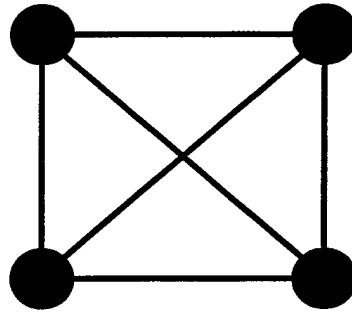
An optimal presentation of a generic graph is based on aesthetic criteria. Aesthetic criteria attempt to characterize readability by means of general optimization goals. These goals consist of:

- 1. Minimize crossing**
- 2. Minimize area**
- 3. Minimize bends (in orthogonal drawings)**
- 4. Minimize slopes (in polyline drawings)**
- 5. Maximize smallest slope**
- 6. Maximize display of symmetries**

In general, one cannot simultaneously optimize two aesthetic criteria. A simple example of this problem can be shown in Figure 2.4. A simple graph of four nodes and six links is displayed with different criteria. Since these criteria cannot optimize all aesthetic criteria, a set of constraints is usually provided as additional input to a graph drawing algorithm. These constraints specify which criteria are more important for the desired layout.



A. Minimize crossing



B. Maximize symmetries

Figure 2.4: Differences in optimization with different criteria.

By allowing a blending of different criteria the user can automatically change the appearance of the display by playing with the constraints. This leads to many complexity issues. Testing planarity takes linear time. Testing upward planarity is NP-hard. Minimizing crossing is NP-hard. All of these constraints add to the complexity of the algorithms. They also add to the time taken to compute a layout.

III. ANALYSIS OF AUTOMATED LAYOUT TECHNIQUES

A. GENERAL

Numerous algorithms exist for automatically laying out graphs. Since the optimal layout for a graph is a NP Complete problem, these algorithms are only approximations to an optimal layout. Almost every algorithm returns a different solution to the question for a specific graph. Some algorithms give different answers when called repeatedly. Usually, a library of algorithms are given to the user to interactively select a layout by running different algorithms until he finds a layout that is aesthetically pleasing and more coherent.

B. GENERAL TECHNIQUES

In general, all algorithmic approaches to laying out a graph in an aesthetically pleasing manner can be broken down into categories. Inside each category there are many variations of the general theme. This section will outline the general themes of these main categories.

Trees or rooted trees are often used to represent hierarchies such as family trees, organizational charts, and search trees. Planar straight-line drawings and orthogonal polyline drawings are commonly used to represent rooted trees. In general, vertices are placed along horizontal lines according to their level. There is a minimum separation distance between two consecutive vertices on the same level. The width of the drawing is as small as possible. Also, for binary trees, left and right children of each vertex are placed to the left and right of the vertex, respectively.

In an inclusion representation of a tree, boxes represent nodes and parent-child relationships are represented by inclusion of one box in another. The tip-over convention is similar to the classical tree graph, however, children of some nodes may be arranged vertically rather than horizontally.

Free trees do not represent hierarchies and have no specific root. Selecting at random a root and then applying an algorithm for a rooted tree works adequately.

Straight-line drawing calls for minimizing crossing of lines in a general graph while minimizing the space required to display the graph and using only straight lines between edges, no polylines. The best approaches to this problem, to date, are heuristic based on a physical model.

The spring embedder algorithm takes such an approach. The drawing process is to simulate a mechanical system, where vertices are replaced by rings, and edges are replaced by springs. The springs attract the rings if they are too far apart, and repel them if they are too close. Variations to this model modify the energy function of the springs by criteria other than just distance.

Planarization involves ensuring that a general graph is planar. If not, it attempts to planarize the graph. This allows many techniques that have been developed for planar graphs to be used.

The most common planarization operation is edge deletion. The smallest set of edges whose deletion yields a planar graph is found. This is equivalent to finding a planar subgraph with a large number of edges. Most of the algorithms that use edge deletion use different approaches for finding a maximum planar subgraph.

Another technique for planarization is splitting. The splitting operation is to make two copies of a vertex and share the neighbors between the two copies. Algorithms that use splitting try to optimize the finding of a minimum splitting sequence.

C. ALGORITHM SEARCH

Relatively little information on graph layout algorithms is available. Only through a search on the Internet was the needed information found. There exist a few home pages on the Internet that deal with automatic layout algorithms. Books on the subject are just emerging.

D. THE GRAPH DRAWING SERVER (GDS)

The graph drawing server is located via Dr. Tamassia's home page at <http://www.cs.brown.edu/people/rt>. This server is a collection of graph drawing algorithms that can be access via a Java applet that can run locally. Basically, the user interface is in Java, and the algorithms are running on the host machine. This doesn't allow for easy inclusion, but it does provide an excellent vehicle to ascertain the abilities of graph drawing algorithms. The following is an explanation of the algorithms available in GDS.

1. Giotto.

Giotto constructs an orthogonal drawing of a graph using a network flow method in the orthogonalization phase to obtain the minimum number of bends. Giotto accepts a general multigraph as input and augments it to create a connected graph. The connected graph is planarized. Vertices with degree greater than four are expanded into rectangular symbols, which are viewed as cycles of degree four to yield a graph with maximum degree four. Finally, the graph is passed through orthogonalization and compaction phases. The time complexity is $O((N+C)^2 \log(N+C))$ where N is the number of vertices in the input graph and C is the number of crossings in the drawing constructed [TAMASSIA97].

2. Giotto with labels.

This is a version of Giotto, which draws each vertex as an expanded box large enough to fit its label.

3. Bend-Stretch.

Bend-Stretch has the same three steps – planarization, orthogonalization, and compaction – as Giotto, and differs only in the method used in the orthogonalization step. It adopts the “bend-stretching” heuristic of the Tamassia and Tollis that only guarantees a constant number of bends on each edge, but runs in linear time. The time complexity is $O((N+C)^2 \log(N+C))$ where N is the number of vertices in the input graph and C is the number of crossings in the drawing constructed [TAMASSIA97].

4. Pairs.

Pairs accepts a general multigraph as input, which is augmented to produce a connected graph and then further augmented to produce a biconnected graph. This biconnected graph is then drawn according to its orthogonal drawing algorithm. The edges added by the augmentation steps are not displayed in the final drawing. The time complexity is $O((N+M) \log(N+M))$ where N is the number of vertices in the input graph and M is the number of edges [TAMASSIA97].

5. Series Parallel Drawing.

This algorithm recognizes a series parallel digraph and constructs an upward drawing of it using the delta-drawing algorithm. This is an implementation of “The Recognition of Series Parallel Digraphs” by Valdes, Tarjan, and Lawler. Any graph is accepted as input; an error message will be displayed if it is not a series parallel digraph [TAMASSIA97].

6. Sugiyama.

Sugiyama constructs a hierarchical drawing of a directed graph according to its algorithm. If the input graph is not directed, it is first converted to a directed graph.

The algorithm uses a three-step process: first, in a layering step, it assigns vertices to horizontal layers. Next, in a crossing-minimization step, it permutes the vertices within the same layer to reduce edge-crossings. Finally, in a bend-reduction step, it readjusts the position of vertices within each layer to reduce edge-bends [TAMASSIA97].

7. Column.

Column is similar to Pairs and differs from it only in the method to optimize the number of bends, rows, and columns used in the drawing, once an st-numbering has been computed. The method used is the one of Biedl and Kant. The time complexity is $O(N+M)$ where N is the number of vertices in the input graph and M is the number of edges [TAMASSIA97].

8. Ortho Upward.

Ortho Upward is an algorithm that produces a straight-line orthogonal upward drawing of a binary tree.

9. Ortho Non-Upward.

Ortho Non-Upward produces a straight-line orthogonal (non-upward) drawing of a binary tree.

10. Planarizer.

Planarizer is the planarization step of Giotto and constructs a planar embedding of the input graph by replacing edge crossing with fictitious vertices. It has time complexity $O((N+C)^2 \log(N+C))$ where N is the number of vertices in the input graph and C is the number of crossings in the drawing constructed [TAMASSIA97].

E. GRAPHLET

Graphlet is a toolkit for graph drawing algorithms. Most applications start with an abstract graph structure that has no coordinates. Arranging the nodes and edges in a nice fashion is a tedious process for humans. Graph drawing algorithms help users to draw graphs. Graphlet's editor toolkit is implemented with C++, LEDA, Tcl/Tk and Graphscript. Below are some outputs of Graphlet [HIMSOLT97].

Graphlet is available on multiple machines and operating systems. It is available on a PC for Windows or Linux. It runs on Sun's running SunOS or Solaris. Graphlet also runs on Hewlett Packard (HP)'s running HP-UX.

With the diversity of platforms, its intuitive user interface, and the source being C++ Graphlet was chosen to display the graphs since the graphical editor was not available.

1. Random Layout.

Random Layout generates random $[x,y]$ positions for the nodes of a graph. It is useful when evaluating different graph algorithms.

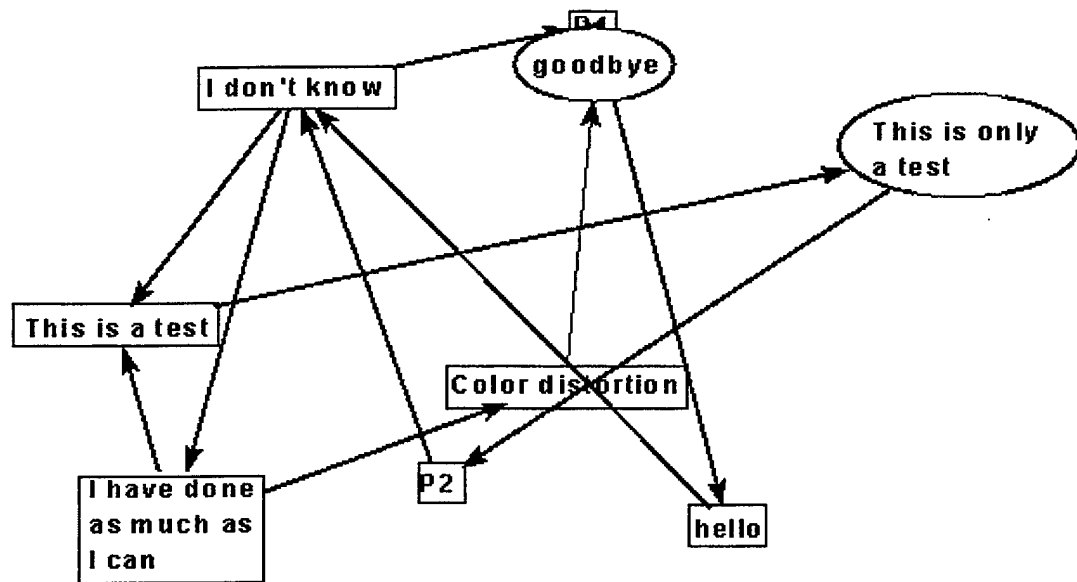


Figure 3.1: Random Layout

2. Spring Embedder with Constraints

Spring Embedder with Constraints is a straight line layout algorithm that tries to maximize space, edge crossing, and angular resolution.

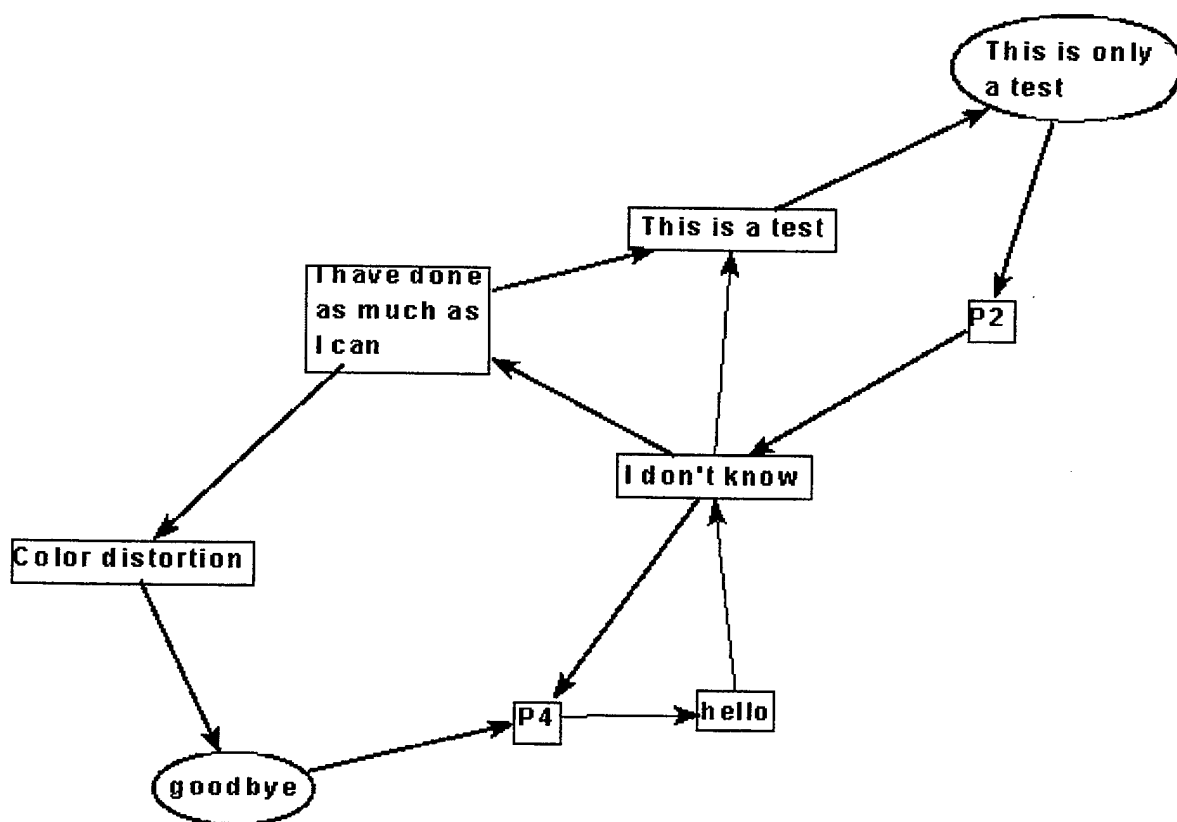


Figure 3.2: Spring Embedder with Constraints

3. Iterative Constraint Spring Embedder

Iterative Constraint Spring Embedder is similar to Spring Embedder with Constraints. However, it iterates the method to produce an orthogonal graph.

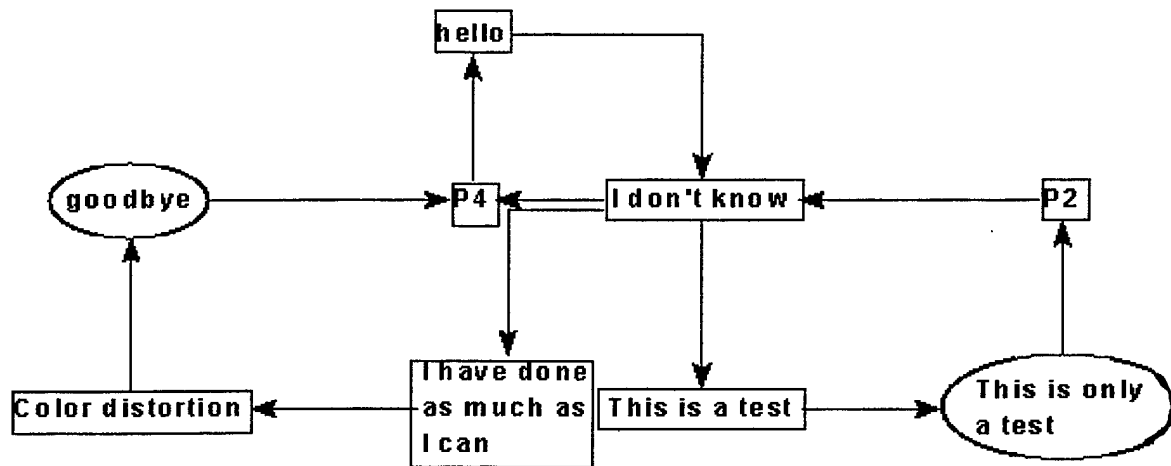


Figure 3.3: Interactive Constraint Spring Embedder

4. Spring Embedder (GEM)

GEM is another Spring Embedder. It seeks to minimize space and edge crossing.

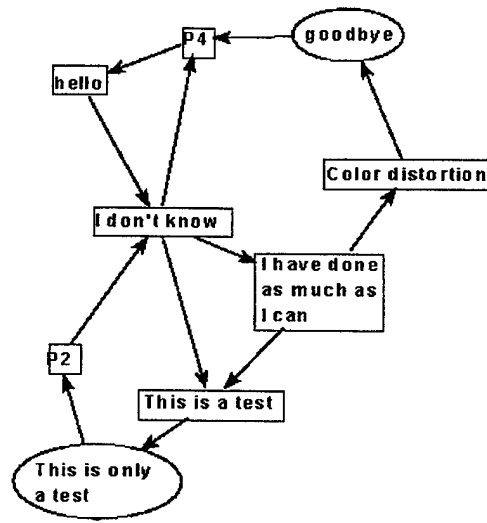


Figure 3.4: Spring Embedder (GEM)

5. Spring Embedder (Kamada)

Kamada is another variation of the Spring Embedder approach.

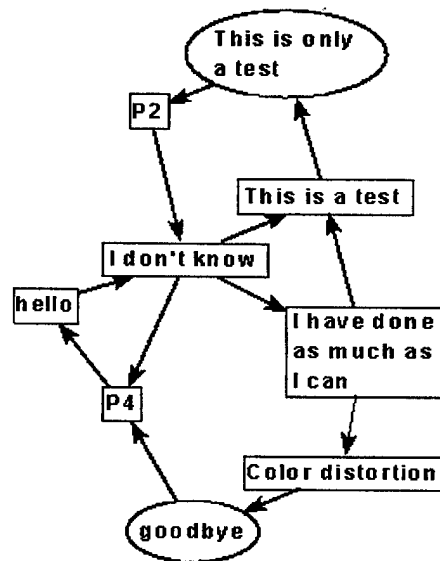


Figure 3.5: Spring Embedder (Kamada)

6. General Graphs (Tunkelang)

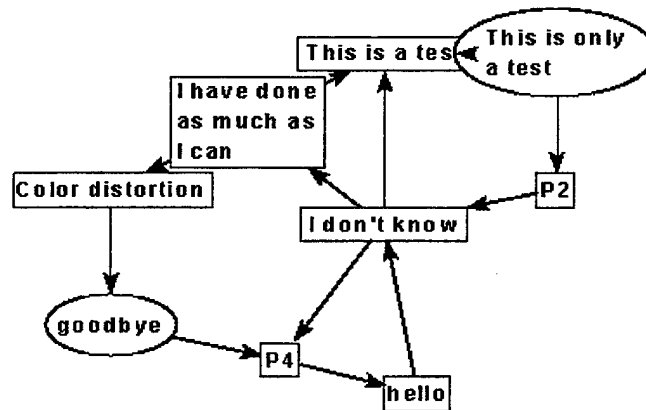


Figure 3.6: General Graphs (Tunkelang)

7. DAG

DAG is a very interesting drawing algorithm. It allows for multiple line segments for edges. It also tries to orient the graph to flow down.

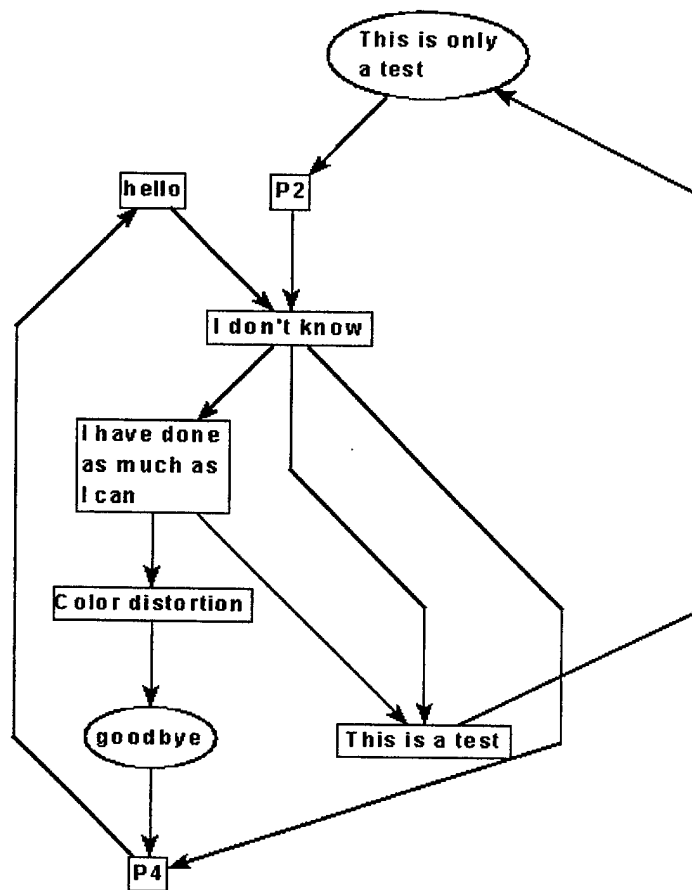


Figure 3.7: DAG

IV. ANALYSIS OF CAPS GRAPHICAL EDITOR

A. GENERAL

The Computer Aided Prototyping System (CAPS) is a software-engineering tool designed to take the drudgery out of prototyping systems. A software developer can design a software system, test the design, and add timing requirements. Most projects become an iterative process of re-visiting the design, making the necessary changes to fix problems, and then testing the new design.

After a few iterations, the graphical display of the prototyped system can become very difficult to understand, since the graphical display requires the user to layout the design. Fixing the design layout by hand is tedious. In the last release of CAPS, Version 1.1, the graphical editor was almost impossible to use when changing the layout to a more meaningful form. The refresh rates made moving a node so cumbersome that the software engineer often didn't want to update a designs layout.

B. CAPS CHARACTERISTICS

The set of graphs that CAPS uses is a subset of all graphs. The following are characteristics that CAPS graphs will have. These characteristics are important in selecting layout algorithms. The most important distinction from the set of all graphs to the set of all graphs that CAPS can create is that all CAPS graphs are directed graphs.

The edges in CAPS represent streams of data. They always flow from a source to a destination. The edges are drawn with arrows to depict direction of the data flow. These edges are splines. Text on the edge represents the variable name in the CAPS PSDL program. CAPS displays one edge per stream. If a call from one object to another requires ten streams, then ten edges are drawn with CAPS graphical editor.

Nodes are represented by two classes of objects with each having a subclass resulting in four distinct node representations. Operators are represented with an oval and terminators are represented with a rectangle. Each of these types can have subgraphs.

This is displayed by putting a double oval or rectangle instead of a single line. Basically, the object is halloed if a subgraph exists. Nodes can be color coded to further differentiate the class.

V. FUTURE REQUIREMENTS OF AUTOMATED LAYOUT TECHNIQUES FOR DIRECTED GRAPHS

A. GENERAL

Research in the area of automated layout techniques for a directed graph is going on in the world, today. Since graphs allow users to visualize many different problems, the ability to display these graphs in an aesthetically pleasing manner is highly desirable. Every year, since 1992, an annual workshop on graph drawing [GD 92, 93, 94, 95, 96, 97] is held to improve the capabilities of existing and new algorithms.

B. RESEARCH AREAS

There exist many research areas in the field of graph drawing. Because the problem in general is NP-Complete, practical algorithms can only be approximations of the ideal solution. Also, the ideal solution can vary depending on the problem area being visualized.

Much research is being conducted in general areas of graph drawing and in application specific areas. Some of the areas that still require future research that will be of particular interest for CAPS are detailed below.

1. Performance Bounds for Planarization.

Although crossing minimization is a fundamental issue, non-trivial performance bounds have not been found for any heuristic. A guaranteed heuristic would be very important both for aesthetic graph drawing and VLSI layout [TAMASSIA94].

2. Simple Planarity Testing.

The known planarity algorithms that achieve linear time complexity are all difficult to understand and implement. This is a serious limitation for their use in practical systems. A simple and efficient algorithm for testing the planarity of a graph and constructing planar representations would be a significant contribution [TAMASSIA94].

3. General Strategy for Straight-Line Drawings.

General strategies have been successfully developed for hierarchical drawings and orthogonal grid drawings. These techniques take several aesthetics into account. The simplicity of straight-line drawing is very appealing, and a general straight-line drawing technique would find immediate applications [TAMASSIA94].

4. Dynamic Drawing Algorithms.

Several graph manipulation systems allow the user to interactively modify a graph by inserting and deleting vertices and edges. Data structures that allow for fast restructuring of the drawing would be very useful. The time needed to re-compute the layout must be small to keep the system from becoming cumbersome, since the algorithm would be called every time an update occurred [TAMASSIA94].

5. Complexity of Bend Minimization.

Several issues on the computational complexity of minimizing bends in planar orthogonal drawings are open. If the embedding is fixed, bend minimization can be done in time $O(n^2 \log n)$. It would be interesting to improve on the sequential complexity and to develop a fast parallel algorithm for the fixed-embedding problem [TAMASSIA94].

6. Angular Resolution of Planar Straight-Line Drawings.

The angular resolution of a planar straight-line drawing is the minimum angle formed by two edges incident on the same vertex. It has been shown that a planar graph of degree d has a drawing with angular resolution upper bound of $O(1/d)$ [TAMASSIA94].

VI. NEW AUTOMATED LAYOUT TECHNIQUES FOR CAPS DESIGN AND IMPLEMENTATION

A. GENERAL

The general idea is to add automated layout techniques to the CAPS graphical editor. However, this is easier said than done. Automated techniques for laying out a graph are really just estimates of a nice layout. A user will still have to make minor adjustments to the graph to fix any minor layout problems. Also, there exist many layout techniques that work best depending on the how the user views certain aesthetic characteristics and the type of graph being displayed.

B. CURRENT PROBLEMS

There are problems related to design and implementation of automated layout techniques for the CAPS graphical editor. The first problem is that the CAPS system is in a state of transition. The PSDL editor is being re-written to use a new PSDL data type. There doesn't exist any graphical editor for the new PSDL data type. Secondly, the graph part of a PSDL data type is private. Without either modifying the PSDL data type to export its layout or making the graph functions in the PSDL data type public, there would be no way to implement new techniques via the PSDL data type.

The use of twin lines in CAPS also creates aesthetic problems. Twin lines are defined as an edge with the same start node and finish node with equal direction. By allowing the user to collapse twin lines for aesthetic reasons, a more easily understood graph is presented. Figure 6.1 demonstrates this point nicely.

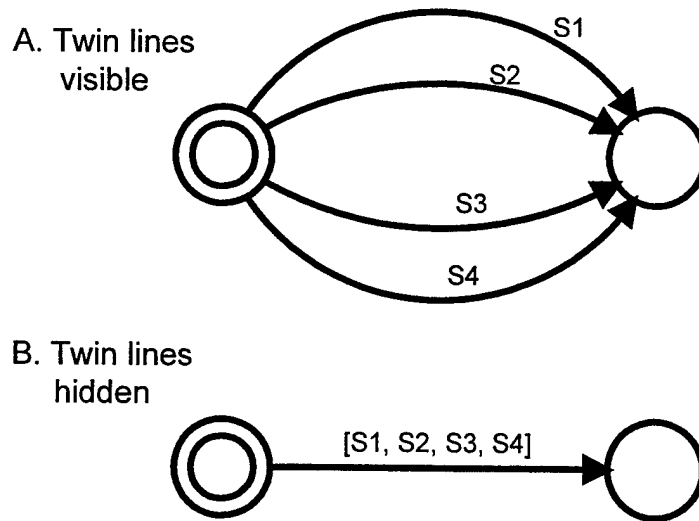


Figure 6.1: Hiding twin lines

C. DESIGN OF TEST SYSTEM

The design of this implementation is not optimal. However, given the availability of the graphical editor in the CAPS system it represents a valid workaround.

Basically, the design is to hook two systems together via some middleware. Since the CAPS system outputs a PSDL file, the file could be read and the graph extracted. This extracted graph would then be fed into a graphical editor that could display the graph. The user interface of this graphical editor would allow the user to run different algorithms and modify the parameters of these algorithms. Figure 6.2 gives a graphical view of this design.

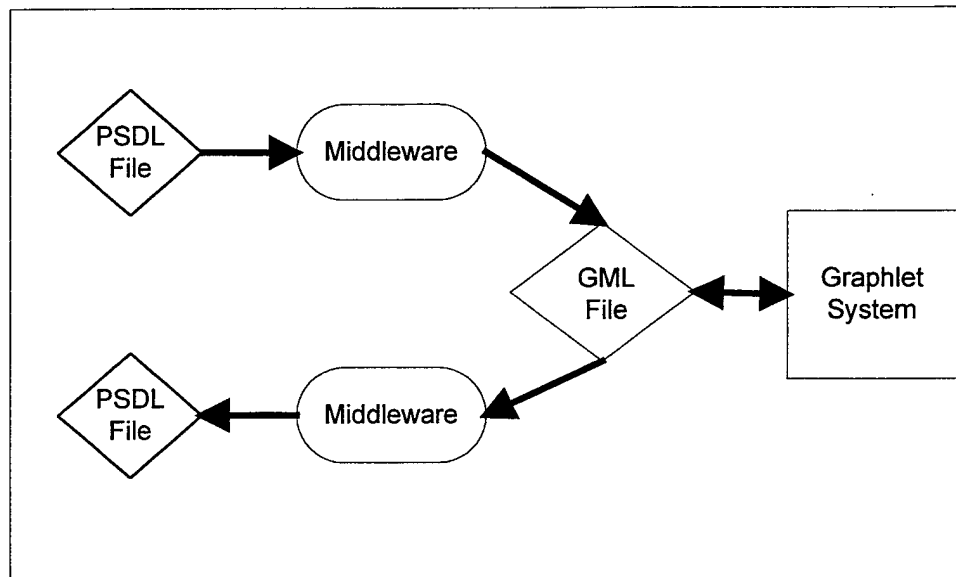


Figure 6.2: Design of test system

D. IMPLEMENTATION OF TEST SYSTEM

1. PSDL Data Type.

The PSDL data type is stored in a PSDL file. These files usually end with the suffix “.psdl”. Since the graph portion of the PSDL data type is private, one must access the graph from a program that uses the PSDL data type without modifying the PSDL data type. Public operators available to a user of the PSDL data type do not allow access of the graph, therefore only by modifying the PSDL data type can new capabilities be added.

2. Middleware.

The middleware is a program that takes a PSDL file and exports a GML file or takes a GML file and updates a PSDL file with the GML file. An example would be the following commands:

```
% middleware -p <psdl file> <gml file>
```

```
% middleware -g <psdl file> <gml file>
```

3. GML File.

The GML File format is a common format to represent a graph used in many graph theory systems. By using this file format, multiple systems could be brought to the table to aid in the drawing of graphs. Also, new algorithms could more easily be integrated into the system. The complete file format for GML is located in document [HIMSOLT96].

4. Graphlet system.

The Graphlet system is freeware that allows a GML file to be read into the system. After the graph is input, the user can choose various algorithms, make slight modifications to the variables in each algorithm. A user can then make minor adjustments to the graph. The new graph can then be saved with the new layout [HIMSOLT97].

E. DESIGN OF REAL SYSTEM

In the last month of this writing, a new version of the CAPS graphical editor has been available. Since the ideal approach is to connect the layout algorithms directly to the graphical editor, the design has changed. Figure 6.3 gives a graphical view of the design for this implementation.

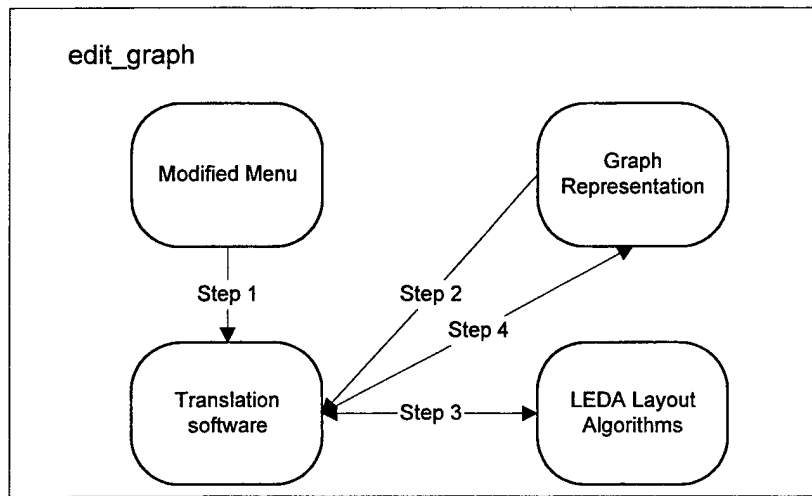


Figure 6.3: Design of REAL system

F. IMPLEMENTATION OF REAL SYSTEM

The implementation of the real system design consisted of porting the LEDA libraries to SunOS 4.1.3. After this was accomplished, the C++ LEDA libraries could be linked with the `edit_graph` program. `Edit_graph` is the executable program for the graphical editor.

The actual linking proved to be more complex than anticipated. CAPS actually uses different compilers for different pieces of the system. It is mostly a mixture of Ada95, C, and C++. LEDA actually needs to be compiled with a newer version of the GNU C++ compiler that CAPS does. In order to link the software, you need to use the linker in the newer version of the GNU C++ compiler.

Two parts of the `edit_graph` need to be modified to use the layout algorithms in LEDA. The first part deals with adding menu items to the system that allow a operator to invoke different algorithms. The second half deals with converting the PSDL data type representation of a graph to a LEDA representation and vis versa.

VII. CONCLUSION AND FUTURE RESEARCH

A. CONCLUSION

The need for automated layout techniques in CAPS is a very real one. These techniques allow a user to clean up the mess associated with iterative edits of the graph. The real place to put these algorithms is inside the graphical editor of the CAPS system. They should be initiated only by user interaction.

Since the graphical editor of the CAPS system was not available for the new PSDL data type, this could not be done. Instead, middleware was developed to allow the two systems to work together.

The graph part of the PSDL data type should use a standard format consistent with the graphing community. This would allow for easy synchronization with leading researchers in topology.

B. FUTURE RESEARCH

Future research needs to be done on the automated layout of graphs that does a better job of placing edge labels. This problem has not been addressed in much detail to date. Currently, most algorithms do reasonably well laying out the nodes, node labels, and edges. Once these algorithms are finished, they could be modified to handle edge labels better. CAPS graphs contain edge labels, operator labels, and maximum execution time (MET) labels. Optimizing the layout with all of these objects would increase the complexity of the algorithm.

The use of twin lines in the CAPS system can become very cluttered in the graphical display. Allowing the user to toggle the twins to visible or hidden would greatly improve the aesthetics of the graph.

The use of splines for edges needs to be studied. Straight line drawing and polyline drawing of edges is the norm. Splines are similar to polyline drawings with the corners rounded. If splines do not aid in the comprehension of the graph, then they

should be replaced with polylines. Drawing a polyline is much faster than drawing a spline. Regardless, the user should be able to choose between splines, straight lines, and polylines to represent edges.

C. CAPS IMPROVEMENT

1. Spline Drawing

Splines are currently drawn one point at a time. The actual algorithm is very inefficient. It doesn't take into account the resolution of the system's display. Duplicate points are very possible. This means that the same point will be drawn more than once. By computing all the points first, then removing duplicate point, a faster algorithm will result.

Streams are displayed as bold splines. For each of the points in the spline, eight additional points are drawn. Basically, all of the original point's neighbors are also drawn point by point. By drawing the bold spline with a filled square of 3x3 pixels instead of 9 separate, one pixel draws, the algorithm would be more efficient.

2. File format

The current format for the PSDL graph is unique to CAPS. If more conventional graph representations were incorporated, then PSDL graphs could be exported to other graph systems. This would also allow for improvements to CAPS graphical editor with minimal changes by using a standard commonly recognized in the graphing community.

APPENDIX A: LEDA LICENSE INFORMATION

A. LEDA LICENSE

You are installing the RESEARCH version (LEDA-R) of LEDA that can be used free of charge for academic research and teaching.

FOR ANY COMMERCIAL USE OF THIS SOFTWARE A LICENSE IS REQUIRED. ANY KIND OF USE BY A COMPANY OR OTHER NON-ACADEMIC INSTITUTION IS CONSIDERED TO BE COMMERCIAL USE. YOU ARE BREAKING A LAW WHEN USING LEDA COMMERCIALY WITHOUT OWNING A LICENSE.

These terms are valid for all LEDA versions following version 3.0. For more information about the license terms please contact:

LEDA Software GmbH
Postfach 151101
66041 Saarbruecken
Germany
email:leda@mpi-sb.mpg.de
fax: +49 681 842502

You are allowed to continue with the installation of LEDA only if you are owner of a valid license or if you intend to use LEDA for academic research or teaching. Otherwise, you must stop the installation now.

B. LEDA INFORMATION

1. LEDA-R-3.5.1

The new LEDA version available on our ftp server is "LEDA-R-3.5.1". The "R" stands for research and has been added to make it distinguishable from the commercial version distributed by LEDA Software GmbH. Please read the Changes files for information about new features and other changes.

LEDA-R-3.5.1 can be used free of charge in academic research and teaching. Licenses for the commercial version "LEDA 3.5.1" are distributed by the LEDA Software GmbH.

2. Differences between LEDA and LEDA-R

The only difference between LEDA-R and LEDA is that the research version may contain additional data types, algorithms or other features which are:

- experimental (not tested enough or not working on all platforms)
- of interest only for particular research
- temporary (may be removed or changed in future versions)

The commercial version will never contain data types, algorithms or features of this kind. It is supposed to stay backward consistent in the sense that old programs should work with new versions of LEDA.

3. Who needs a license?

- Every organization that uses LEDA and is not an academic research institute or school.
- Everyone who sells programs that are developed using LEDA.
- Everyone who delivers programs that are developed using LEDA to non-academic organizations.

Holders of a licence for the commercial version are free to also use the research version for commercial use.

Please write to leda@mpi-sb.mpg.de if you want to

- * send bug reports or suggestions
- * get information on the commercial license
- * be on our mailing list

Subscribe to the LEDA newsgroup comp.lang.c++.leda

C. LEDA SOURCE CODE

```
/******
*****
+
+ LEDA 3.5.1
+
+ _g_array.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****
*****/
#include <LEDA/graph.h>

//-----
//
// graph maps and arrays
//
// graph_map: base of node/edge/face_map/array
//
// by S. Naeher (1995,1996)
//-----

graph_map::~graph_map()
{ if (g && g_index != 0) g->unregister_map(this);
  if (table) delete[] table;
}

int graph_map::next_power(int s) const
{ if (s==0) return 0;
  int p = 1;
  while (p < s) p <<= 1;
  return p;
}

void graph_map::re_init_entry(node v)
{ if (g_index > -1)
  { init_entry(v->data[g_index]);
  }
  else
  { int i = index(v);
    if (i < table_size)
    { clear_entry(table[i]);
      init_entry(table[i]);
    }
  }
}
```

```

void graph_map::re_init_entry(edge e)
{ if (g_index > -1)
    init_entry(e->data[g_index]);
  else
    { int i = index(e);
      if (i < table_size)
        { clear_entry(table[i]);
          init_entry(table[i]);
        }
    }
}

void graph_map::re_init_entry(face f)
{ if (g_index > -1)
    init_entry(f->data[g_index]);
  else
    { int i = index(f);
      if (i < table_size)
        { clear_entry(table[i]);
          init_entry(table[i]);
        }
    }
}

void graph_map::init_table(GenPtr* start, GenPtr* stop)
{ if (g_index == -1)
    for(GenPtr* q=start; q < stop; q++) init_entry(*q);
  else
    if (g && g_index > 0)
      {
        switch (kind) {
          case 0 : { node v;
                    forall_nodes(v,*g) init_entry(v->data[g_index]);
                    break;
                  }
          case 1 : { edge e;
                    forall_edges(e,*g) init_entry(e->data[g_index]);
                    break;
                  }
          case 2 : { face f;
                    forall_faces(f,*g) init_entry(f->data[g_index]);
                    break;
                  }
        }
      }
}

void graph_map::clear_table()
{ if (g_index == -1)
    { GenPtr* stop = table + table_size;
      for(GenPtr* q=table; q < stop; q++) clear_entry(*q);
    }
  else
    if (g && g_index > 0)
      { switch (kind) {

```

```

        case 0 : { node v;
                    forall_nodes(v,*g) clear_entry(v->data[g_index]);
                    break;
                }
        case 1 : { edge e;
                    forall_edges(e,*g) clear_entry(e->data[g_index]);
                    break;
                }
        case 2 : { face f;
                    forall_faces(f,*g) clear_entry(f->data[g_index]);
                    break;
                }
    }
}

void graph_map::resize_table(int sz)
{
    GenPtr* old_table = table;
    GenPtr* old_stop = table + table_size;

    table_size = sz;
    table = new GenPtr[sz];
    if (table == 0) error_handler(1, " graph_map: out of memory");

    GenPtr* p = old_table;
    GenPtr* q = table;
    while (p < old_stop) *q++ = *p++;

    init_table(q,table+sz);

    if (old_table != old_stop) delete[] old_table;
}

void graph_map::init(const graph* G, int sz, int k)
{
    if (g != G)
    { if (g && g_index != 0) g->unregister_map(this);
      kind = k;
      g = (graph*)G;
      if (g) g_index = g->register_map(this);
    }

    if (g_index > -1)
    { table = 0;
      table_size = 0;
      return;
    }

    clear_table();
    if (table_size > 0) delete[] table;

    table = 0;

    table_size = next_power(sz);
    if (table_size > 0)
    { table = new GenPtr[table_size];
      if (table == 0) error_handler(1, " graph_map: out of memory");
    }
}

```

```

}

graph_map::graph_map(const graph* G, int k)
{ kind = k;
  g = (graph*)G;
  g_index = 0;
  table = 0;
  table_size = 0;
}

graph_map::graph_map(const graph* G, int sz, int k)
{
  kind = k;
  g = (graph*)G;
  g_index = -1;

  if (g) g_index = g->register_map(this);

  if (g_index > -1)
  { table = 0;
    table_size = 0;
    return;
  }

  def_entry = 0;
  table = 0;
  table_size = next_power(sz);
  if (table_size > 0)
  { table = new GenPtr[table_size];
    if (table == 0) error_handler(1, "graph_map: out of memory");
  }
}

graph_map::graph_map(const graph_map& M)
{ kind = M.kind;
  g = M.g;
  if (M.g_index == 0)
  { g_index = 0;
    table = 0;
    return;
  }
  g_index = -1;
  if (g) g_index = g->register_map(this);
  def_entry = 0;
  table = 0;
  table_size = M.table_size;
  if (table_size > 0)
  { table = new GenPtr[table_size];
    if (table == 0) error_handler(1, "graph_map: out of memory");
    GenPtr* p = table;
    GenPtr* stop = M.table+M.table_size;
    for(GenPtr* q=M.table; q < stop; q++)
    { *p = *q;
      M.copy_entry(*p);
      p++;
    }
  }
}

```



```

graph_map& graph_map::operator=(const graph_map& M)
{ if (&M == this) return *this;
  clear_table();
  if (table_size > 0) delete[] table;
  if (g && g_index != 0) g->unregister_map(this);
  table = 0;
  kind = M.kind;
  g = M.g;
  if (M.g_index == 0)
  { g_index = 0;
    table = 0;
    return *this;
  }
  g_index = -1;
  if (g) g_index = g->register_map(this);
  table_size = M.table_size;
  if (table_size > 0)
  { table = new GenPtr[table_size];
    if (table == 0) error_handler(1, " graph_map: out of memory");
    GenPtr* p = table;
    GenPtr* stop = M.table+M.table_size;
    for(GenPtr* q=M.table; q < stop; q++)
    { *p = *q;
      copy_entry(*p);
      p++;
    }
  }
  return *this;
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _g_generate.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
+ *****/
#include <LEDA/graph.h>
#include <LEDA/ugraph.h>
#include <LEDA/vector.h>
#include <LEDA/matrix.h>
#include <LEDA/array2.h>

#include <ctype.h>
#include <math.h>

//-----
// some graph generators
//
// S. Naeher (1995-1996)
//-----

void complete_graph(graph& G, int n, bool directed)
{
    G.clear();

    node* V = new node[n];

    for(int i=0;i<n;i++) V[i] = G.new_node();

    if (directed)
        { //memory_allocate_block(sizeof(node_struct),n);
          for(int i=0;i<n;i++)
              for(int j=0;j<n;j++) G.new_edge(V[i],V[j]);
        }
    else
        { //memory_allocate_block(sizeof(edge_struct),n*n/2);
          for(int i=0; i<n; i++)
              for(int j=i+1; j<n; j++) G.new_edge(V[i],V[j]);
        }

    delete[] V;
}

void grid_graph(graph& G, int n)

```

```

{ node_array<double> xcoord;
  node_array<double> ycoord;
  grid_graph(G,xcoord,ycoord,n);
}

void grid_graph(graph& G, node_array<double>& xcoord,
               node_array<double>& ycoord, int n)
{
  array2<node> A(n,n);
  node v;
  int N = n*n;
  int x;
  int y;

  double d = 1.0/(n+1);

  G.clear();

  xcoord.init(G,N,0);
  ycoord.init(G,N,0);

  for(y=0; y<n; y++)
    for(x=0; x<n; x++)
      { A(x,y) = v = G.new_node();
        xcoord[v] = (x+1)*d;
        ycoord[v] = (y+1)*d;
      }

  for(x=0; x<n; x++)
    for(y=0; y<n; y++)
      { if (x < n-1) G.new_edge(A(x,y),A(x+1,y));
        if (y < n-1) G.new_edge(A(x,y),A(x,y+1));
      }
}

void complete_bigraph(graph& G, int n1, int n2, list<node>& A,
list<node>& B)
{
  G.clear();

  while (n1--) A.append(G.new_node());
  while (n2--) B.append(G.new_node());

  list_item a,b;
  forall_items(a,A)
    forall_items(b,B)
      G.new_edge(A[a],B[b]);
}

void user_graph(graph& G)
{ int n = read_int("|V| = ");
  int i,j;

  node* V = new node[n];
  for(j=0; j<n; j++) V[j] = G.new_node();

  for(j=0; j<n; j++)
  { list<int> il;
    int ok = false;
    while (!ok)

```

```

    { ok = true;
      cout << "edges from [" << j << "]" to: ";
      il.read();
      forall(i,il)
        if (i < 0 || i >= n)
          { ok=false;
            cout << "illegal node " << i << "\n";
          }
      }
      forall(i,il) G.new_edge(V[j],V[i]);
    }
  G.print();
  if (Yes("save graph ? ")) G.write(read_string("file: "));
  delete[] V;
}

void test_graph(graph& G)
{
  G.clear();
  char c;

  do c = read_char("graph: f(file) r(random) c(omplete) p(lanar) u(ser):
");
  while (c!='f' && c!='r' && c!='c' && c!='p'&& c!='u');

  switch (c) {

    case 'f' : { G.read(read_string("file: "));
                 break;
               }

    case 'u' : { user_graph(G);
                 break;
               }

    case 'c' : { complete_graph(G,read_int("|V| = "));
                 break;
               }

    case 'r' : { int n = read_int("|V| = ");
                 int m = read_int("|E| = ");
                 random_graph(G,n,m);
                 break;
               }

    case 'p' : { random_planar_graph(G,read_int("|V| = "));
                 break;
               }

    } //switch
}

void test_ugraph(ugraph& G)
{
  G.clear();
  char c;

```

```

do c = read_char("graph: f(file) r(andom) c(omplete) p(lanar) u(ser):
");
while (c!='f' && c!='r' && c!='c' && c!='p' && c!='u');

int i;
node v;

switch (c) {

case 'f' : { G.read(read_string("file: "));
             break;
           }

case 'u' : { int n = read_int("|V| = ");
             int j = 0;
             node* V = new node[n];
             for(i=0; i<n; i++) V[i] = G.new_node();
             forall_nodes(v,G)
             { list<int> il;
               cout << "edges from " << j++ << " to: ";
               il.read();
               forall(i,il)
                 if (i >= 0 && i < n) G.new_edge(v,V[i]);
                 else cerr << "illegal node " << i << "
(ignored)\n";
             }
             G.print();
             if (Yes("save graph ? ")) G.write(read_string("file:
"));
             delete[] V;
             break;
           }

case 'c' : { int n = read_int("|V| = ");
             complete_graph(G,n);
             break;
           }

case 'r' : { int n = read_int("|V| = ");
             int m = read_int("|E| = ");
             random_graph(G,n,m);
             break;
           }

} //switch

}

void test_bigraph(graph& G, list<node>& A, list<node>& B)
{
  int a,b;
  int n1 = 0;
  int n2 = 0;
  char c;

  do c = read_char("bipartite graph: f(file) r(andom) c(omplete) u(ser):
");

```

```

while (c!='f' && c!='r' && c!='c' && c!='u');

A.clear();
B.clear();
G.clear();

if (c!='f')
{ n1 = read_int("|A| = ");
  n2 = read_int("|B| = ");
}

switch (c) {

case 'f' : { G.read(read_string("file: "));
             node v;
             forall_nodes(v,G)
             if (G.outdeg(v) > 0) A.append(v);
             else B.append(v);

             break;
           }

case 'u' : { node* AV = new node[n1+1];
             node* BV = new node[n2+1];

             for(a=1; a<=n1; a++) A.append(AV[a] = G.new_node());
             for(b=1; b<=n2; b++) B.append(BV[b] = G.new_node());

             for(a=1; a<=n1; a++)
             { list<int> il;
               cout << "edges from " << a << " to: ";
               il.read();
               forall(b,il)
               if (b<=n2) G.new_edge(AV[a],BV[b]);
               else break;
               if (b>n2) break;
             }
             delete[] AV;
             delete[] BV;
             break;
           }

case 'c' : complete_bigraph(G,n1,n2,A,B);
             break;

case 'r' : { int m = read_int("|E| = ");
             random_bigraph(G,n1,n2,m,A,B);
             break;
           }

           } // switch
}

void cmdline_graph(graph& G, int argc, char** argv)
{
  // construct graph from cmdline arguments

```

```

    if (argc == 1)                // no arguments
    { test_graph(G);
      return;
    }
    else
    if (argc == 2)                // one argument
    { if (isdigit(argv[1][0]))
      { cout << "complete graph |V| = " << argv[1];
        newline;
        newline;
        complete_graph(G,atoi(argv[1]));
      }
      else
      { cout << "reading graph from file " << argv[1];
        newline;
        newline;
        G.read(argv[1]);
      }
      return;
    }
    else
    if (argc == 3 && isdigit(argv[1][0]) && isdigit(argv[2][0]))
    { cout << "random graph |V| = " << argv[1] << " |E| = " <<
argv[2];
      newline;
      newline;
      random_graph(G,atoi(argv[1]),atoi(argv[2]));
      return;
    }

    error_handler(1,"cmdline_graph: illegal arguments");
}

//-----
// triangulated planar graph
//-----

struct triang_point {

double x;
double y;
node    v;

LEDA_MEMORY(triang_point)

triang_point(double a=0, double b = 0) { x = a; y = b; v = nil; }
triang_point(const triang_point& p)      { x = p.x; y = p.y; v = p.v; }
~triang_point() {};

friend bool right_turn(const triang_point& a, const triang_point& b,
const triang_point& c)
{ return (a.y-b.y)*(a.x-c.x)+(b.x-a.x)*(a.y-c.y) > 0; }

friend bool left_turn(const triang_point& a, const triang_point& b,
const triang_point& c)

```

```

{ return (a.y-b.y)*(a.x-c.x)+(b.x-a.x)*(a.y-c.y) < 0; }

friend bool operator==(const triang_point& a, const triang_point& b)
{ return a.x == b.x && a.y == b.y; }

friend ostream& operator<<(ostream& out, const triang_point& p)
{ return out << p.x << " " << p.y; }

friend istream& operator>>(istream& in, triang_point& p)
{ return in >> p.x >> p.y; }

friend int compare(const triang_point& p, const triang_point& q)
{ int c = compare(p.x,q.x);
  if (c==0) c = compare(p.y,q.y);
  return c;
}

};

void triangulated_planar_graph(graph& G, list<node>& outer_face,
                                node_array<double>& xcoord,
                                node_array<double>& ycoord, int
n)
{
  G.clear();

  list<triang_point> L;

  while(n--)
  { double x = rand_int(0,10000000)/10000000.0;
    double y = rand_int(0,10000000)/10000000.0;
    L.append(triang_point(x,y));
  }

  L.sort(); // sort triang_points lexicographically

  list<triang_point> CH;
  list_item last;
  triang_point p,q;

  // eliminate multiple triang_points

  list_item it;
  forall_items(it,L)
  { list_item it1 = L.succ(it);
    while (it1 != nil && L[it1] == L[it])
    { L.del(it1);
      it1 = L.succ(it);
    }
  }

  n = L.length();

  xcoord.init(G,n,0);
  ycoord.init(G,n,0);

  forall_items(it,L)

```



```

{ node v = G.new_node();
  xcoord[v] = L[it].x;
  ycoord[v] = L[it].y;
  L[it].v = v;
}

// initialize convex hull with first two points

p = L.pop();
CH.append(p);

while (L.head() == p) L.pop();

q = L.pop();
last = CH.append(q);

G.new_edge(p.v, q.v);

// scan remaining points
forall(p, L)
{
    node v = p.v;

    G.new_edge(v, CH[last].v);

    // compute upper tangent (p, up)

    list_item up = last;
    list_item it = CH.cyclic_succ(up);

    while (left_turn(CH[it], CH[up], p))
    { up = it;
      it = CH.cyclic_succ(up);
      G.new_edge(v, CH[up].v);
    }

    // compute lower tangent (p, low)

    list_item low = last;
    it = CH.cyclic_pred(low);

    while (right_turn(CH[it], CH[low], p))
    { low = it;
      it = CH.cyclic_pred(low);
      G.new_edge(v, CH[low].v);
    }

    // remove all points between up and low

    if (up != low)
    { it = CH.cyclic_succ(low);

      while (it != up)
      { CH.del(it);
        it = CH.cyclic_succ(low);
      }
    }
}

```

```

    }
}

// insert new point
last = CH.insert(p,low);

}

outer_face.clear();
forall(p,CH) outer_face.append(p.v);
}

void triangulated_planar_graph(graph& G, int m)
{ node_array<double> xcoord;
  node_array<double> ycoord;
  list<node> L;
  triangulated_planar_graph(G,L,xcoord,ycoord,m);
}

static bool tutte_embed(const graph& G, const node_array<bool>& fixed,
                        node_array<double>& xpos, node_array<double>&
ypos)
{ node v,w;
  edge e;

  list<node> other_nodes;
  forall_nodes(v,G)
    if(!fixed[v]) other_nodes.append(v);

  node_array<int> ind(G);          // position of v in other_nodes and A

  int i = 0;
  forall(v,other_nodes) ind[v] = i++;

  int n = other_nodes.size();      // #other nodes
  vector coord(n);                // coordinates (first x then y)
  vector rhs(n);                  // right hand side
  matrix A(n,n);                  // equations

  // initialize non-zero entries in matrix A
  forall(v,other_nodes)
  {
    double one_over_d = 1.0/double(G.degree(v));
    forall_inout_edges(e,v)
    {
      // get second node of e
      w = (v == source(e)) ? target(e) : source(e);
      if(!fixed[w]) A(ind[v],ind[w]) = one_over_d;
    }
    A(ind[v],ind[v]) = -1;
  }

  if(!A.det()) return false;

  // compute right hand side for x coordinates
  forall(v,other_nodes)
  { rhs[ind[v]] = 0;

```

```

    double one_over_d = 1.0/double(G.degree(v));
    forall_inout_edges(e,v)
    { // get second node of e
        w = (v == source(e)) ? target(e) : source(e);
        if(fixed[w]) rhs[ind[v]] -= (one_over_d*xpos[w]);
    }
}

// compute x coordinates
coord = A.solve(rhs);
forall(v,other_nodes) xpos[v] = coord[ind[v]];

// compute right hand side for y coordinates
forall(v,other_nodes)
{ rhs[ind[v]] = 0;
  double one_over_d = 1.0/double(G.degree(v));
  forall_inout_edges(e,v)
  { // get second node of e
      w = (v == source(e)) ? target(e) : source(e);
      if(fixed[w]) rhs[ind[v]] -= (one_over_d*ypos[w]);
  }
}

// compute y coordinates
coord = A.solve(rhs);
forall(v,other_nodes) ypos[v] = coord[ind[v]];

return true;
}

void triangulated_planar_graph(graph& G, node_array<double>& xcoord,
                                node_array<double>& ycoord, int
n)
{ list<node> L;
  triangulated_planar_graph(G,L,xcoord,ycoord,n);

  if (n > 128) return;

  node_array<bool> fixed(G,false);

  double step = 6.2832/L.length();
  double alpha = 0;
  node v;
  forall(v,L)
  { xcoord[v] = cos(alpha);
    ycoord[v] = sin(alpha);
    alpha+=step;
    fixed[v] = true;
  }

  tutte_embed(G,fixed,xcoord,ycoord);
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _g_gmlio.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
+ *****/
// ----- //
// writing LEDA graphs in GML format //
// reading LEDA graphs in GML format //
// //
// David Alberts (1996) //
// David Alberts (1997) new version, accepts unknown lists, etc. //
// no more lex/yacc //
// ----- //

#include<LEDA/graph.h>
#include<LEDA/stream.h>
#include<LEDA/gml_graph.h>

bool graph::write_gml(string outfile,
                      void (*node_cb)(ostream&,const graph*, const
node),
                      void (*edge_cb)(ostream&,const graph*, const
edge)) const
// writes a graph description in GML format to outfile.
// If an error occurs, false is returned.
{
    file_ostream out(outfile);

    if(out.fail()) return false;
    else return write_gml(out,node_cb,edge_cb);
}

bool graph::write_gml(ostream& out,
                      void (*node_cb)(ostream&,const graph*, const
node),
                      void (*edge_cb)(ostream&,const graph*, const
edge)) const
// writes a graph description in GML format to outfile.
// If an error occurs, false is returned.
{
    if(out.fail()) return false;

    string void_str("void");

    out << "Creator " << "'" << "LEDA write_gml" << "'" << "\n\n";
    out << "graph [\n\n";
    out << " directed " << (is_directed() ? 1 : 0) << "\n";

```

```

out << "\n";

node v;
if((string(node_type()) != void_str) || node_cb)
{
    forall_nodes(v,*this)
    {
        out << "  node [\n";
        out << "    id " << index(v) << "\n";
        if(string(node_type()) != void_str)
        {
            out << "      parameter " << "'" << get_node_entry_string(v);
            out << "'" << "\n";
        }
        if(node_cb) (*node_cb)(out,this,v);
        out << "  ]\n";
    }
}
else
    forall_nodes(v,*this) out << "  node [ id " << index(v) << " ]\n";

out << "\n";

edge e;
if((string(edge_type()) != void_str) || edge_cb)
{
    forall_edges(e,*this)
    {
        out << "  edge [\n";
        out << "    source " << index(source(e)) << "\n";
        out << "    target " << index(target(e)) << "\n";
        if(string(edge_type()) != void_str)
        {
            out << "      parameter " << "'" << get_edge_entry_string(e);
            out << "'" << "\n";
        }
        if(edge_cb) (*edge_cb)(out,this,e);
        out << "  ]\n";
    }
}
else
{
    forall_edges(e,*this)
    {
        out << "  edge [ source " << index(source(e)) << " ";
        out << "target " << index(target(e)) << " ]\n";
    }
}
out << "\n]\n";

return true;
}

bool graph::read_gml(string s)
{
    gml_graph* parser = new gml_graph(*this,s.cstring());
    bool ok = !parser->errors();
    delete parser;
    return ok;
}

```

```
bool graph::read_gml(istream& in)
{
    gml_graph* parser = new gml_graph(*this,in);
    bool ok = !parser->errors();
    delete parser;
    return ok;
}
```

```

/*****
+
+ LEDA 3.5.1
+
+ _g_inout.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****/

//-----
// graph i/o
//
// S. Naeher (last modified: December 1996)
//-----

#include <LEDA/graph.h>
#include <LEDA/stream.h>
#include <ctype.h>
#include <string.h>

const char delim = '|';

void graph::write(string file_name) const
{ ofstream out(file_name);
  if (out.fail())
    error_handler(1,string("graph::write() cannot open file
%s",file_name));
  write(out);
}

void graph::write(ostream& out) const
{
  int* A = new int[max_n_index+1];
  int* B = new int[max_e_index+1];

  for(int i=0; i<= max_e_index; i++) B[i] = 0;

  // nodes get numbers from 1 to |V|

  int n_count = 1;
  int e_count = 1;

  out << "LEDA.GRAPH" << endl;
  out << node_type() << endl;
  out << edge_type() << endl;

```

```

out << v_list.length() << endl;

node v;
forall_nodes(v,*this)
{ out << delim << '{';
  write_node_entry(out,v->data[0]);
  out << '}' << delim << endl;
  A[index(v)] = n_count++;
}

out << number_of_edges() << endl;

forall_nodes(v,*this)
{ edge e;
  int s = A[index(v)];
  forall_adj_edges(e,v)
  { if (source(e) != v) continue; // necessary for ugraphs
    int t = A[index(target(e))];
    int r = (e->rev) ? B[index(e->rev)] : 0;
    out << s << " " << t << " " << r << " " << delim << '{';
    write_edge_entry(out,e->data[0]);
    out << '}' << delim << endl;
    B[index(e)] = e_count++;
  }
}

delete[] A;
delete[] B;

}

int graph::read(string file_name)
{ ifstream in(file_name);
  if (in.fail()) return 1;
  return read(in);
}

static void read_data_entry(istream& in, char* buf, int buf_sz)
{ char* p = buf-1;
  char c;

  do in.get(c); while (isspace(c));
  if (c != delim)
  { in.putback(c);
    string line = read_line(in);
    strcpy(buf,line.cstring());
    return;
  }

  in.get(c);
  if (c != '{')
    error_handler(1,"graph::read(): error in graph format.");

  int nested = 1;

  while (nested)
  { in.get(c);
    if (c == delim && p >= buf)
    { if (*p == '}')
        nested--;
    }
  }
}

```



```

        else
        { if (buf_sz-- <= 0)
            error_handler(1,"graph::read: data overflow");
            *++p = c;
            in.get(c);
            if (c == '{') nested++;
        }
    }

    if (nested)
    { if (buf_sz-- <= 0)
        error_handler(1,"graph::read: data overflow");
        *++p = c;
    }
}

*p = '\0';
}

int graph::read(istream& in)
{
    clear();

    int result = 0;
    int n,i,v,w,r;

    string this_n_type = node_type();
    string this_e_type = edge_type();

    string d_type,n_type,e_type;

    in >> d_type;
    in >> n_type;
    in >> e_type;
    in >> n;

    if (d_type != "LEDA.GRAPH") return 3;

    read_line(in);

    node* A = new node[n+1];

    char data_str[1024];

    if (this_n_type == "void" || n_type != this_n_type) // do not read
node info
    { for (i=1; i<=n; i++)
        { A[i] = new_node();
          read_data_entry(in,data_str,1024);
        }
        if (n_type != this_n_type) result = 2; // incompatible node
types
    }
    else
    if (this_n_type == "string")
    for (i=1; i<=n; i++)
    { A[i] = new_node(0);
      read_data_entry(in,data_str,1024);
      A[i]->data[0] = leda_copy(string(data_str));
    }
}

```

```

    }
    else
        for (i=1; i<=n; i++)
        { A[i] = new_node(0);
          read_data_entry(in,data_str,1024);
          istrstream str_in(data_str,strlen(data_str));
          read_node_entry(str_in,A[i]->data[0]);
        }

    in >> n;          // number of edges
    edge* B = new edge[n+1];

    if (this_e_type == "void" || e_type != this_e_type) // do not read
    edge info
    { if (e_type != this_e_type) result = 2;    // incompatible edge
    types
        for (i=1; i<=n; i++)
        { in >> v >> w >> r;
          edge e = new_edge(A[v],A[w]);
          read_data_entry(in,data_str,1024);
          B[i] = e;
          if (r > 0) set_reversal(e,B[r]);
        }
    }
    else
    if (this_e_type == "string")
        for (i=1; i<=n; i++)
        { in >> v >> w >> r;
          edge e = new_edge(A[v],A[w],GenPtr(0));
          read_data_entry(in,data_str,1024);
          e->data[0] = leda_copy(string(data_str));
          B[i] = e;
          if (r > 0) set_reversal(e,B[r]);
        }
    else
        for (i=1; i<=n; i++)
        { in >> v >> w >> r;
          edge e = new_edge(A[v],A[w],GenPtr(0));
          read_data_entry(in,data_str,1024);
          istrstream str_in(data_str,strlen(data_str));
          read_edge_entry(str_in,e->data[0]);
          B[i] = e;
          if (r > 0) set_reversal(e,B[r]);
        }

    delete[] A;
    delete[] B;

    return result;
}

void graph::print_node(node v,ostream& o) const
{ if (super() != 0)
    super()->print_node(node(graph::inf(v)),o);
    else
    { o << "[" << index(v) << "]" ;
      print_node_entry(o,v->data[0]);
    }
}

```

```

void graph::print_edge(edge e, ostream& o) const
{ if (super() != 0)
    super()->print_edge(edge(graph::inf(e)), o);
  else
  { o <<    "[" << index(source(e)) << "];";
    o <<    ((undirected) ? "==" : "--");
    print_edge_entry(o, e->data[0]);
    o <<    ((undirected) ? "==" : "-->");
    o <<    "[" << index(target(e)) << "];";
  }
}

void graph::print(string s, ostream& out) const
{ node v;
  edge e;
  out << s << endl;
  forall_nodes(v, *this)
  { print_node(v, out);
    out << " : ";
    forall_adj_edges(e, v) print_edge(e, out);
    out << endl;
  }
  out << endl;
}

// convert node and edge entries into a string and vice versa

string graph::get_node_entry_string(node v) const
{ ostream out;
  write_node_entry(out, v->data[0]);
  out << ends;
  char* p = out.str();
  string s(p);
  delete[] p;
  return s;
}

string graph::get_edge_entry_string(edge e) const
{ ostream out;
  write_edge_entry(out, e->data[0]);
  out << ends;
  char* p = out.str();
  string s(p);
  delete[] p;
  return s;
}

void graph::set_node_entry(node v, string s)
{ clear_node_entry(v->data[0]);
  if (strcmp(node_type(), "string") == 0)
    v->data[0] = leda_copy(s);
  else
  { istream in(s.cstring());
    read_node_entry(in, v->data[0]);
  }
}

```

```

void graph::set_edge_entry(edge e, string s)
{ clear_edge_entry(e->data[0]);
  if (strcmp(edge_type(),"string") == 0)
    e->data[0] = leda_copy(s);
  else
    { istringstream in(s.cstring());
      read_edge_entry(in,e->data[0]);
    }
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _g_map.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****/
#include <LEDA/graph.h>

// reversal edges

edge graph::face_cycle_succ(edge e) const
{ return cyclic_adj_pred(reversal(e)); }

edge graph::face_cycle_pred(edge e) const
{ return reversal(cyclic_adj_succ(e)); }

edge graph::succ_face_edge(edge e) const
{ return cyclic_adj_pred(reversal(e)); }

edge graph::pred_face_edge(edge e) const
{ return reversal(cyclic_adj_succ(e)); }

void graph::set_reversal(edge e, edge r)
{ if ( source(e) != target(r) || target(e) != source(r) )
  error_handler(1,"graph::set_reversal: edges are not reversals of each
other");
  e->rev = r;
  r->rev = e;
}

static int map_edge_ord1(const edge& e) { return index(source(e)); }
static int map_edge_ord2(const edge& e) { return index(target(e)); }

bool graph::make_map()
{
  // computes for every edge e = (v,w) in G its reversal r = (w,v)
  // in G ( nil if not present). Returns true if every edge has a
  // reversal and false otherwise.

  int n      = max_node_index();
  int count = 0;

  list<edge> E11 = all_edges();
  list<edge> E12 = E11;

  edge e;

```

```

forall(e, E11) e->rev = 0;

E11.bucket_sort(0,n,&map_edge_ord2);
E11.bucket_sort(0,n,&map_edge_ord1);
E12.bucket_sort(0,n,&map_edge_ord1);
E12.bucket_sort(0,n,&map_edge_ord2);

// merge E11 and E12 to find corresponding edges
while (! E11.empty() && ! E12.empty())
{ edge e = E11.head();
  edge r = E12.head();
  if (target(r) == source(e))
    if (source(r) == target(e))
      { e->rev = r;
        E12.pop();
        E11.pop();
        count++;
      }
    else
      if (index(source(r)) < index(target(e)))
        E12.pop();
      else
        E11.pop();

  else
    if (index(target(r)) < index(source(e)))
      E12.pop();
    else
      E11.pop();

}

return (count == number_of_edges()) ? true : false;
}

void graph::make_map(list<edge>& R)
{ if (make_map()) return;
  list<edge> el = all_edges();
  edge e;
  forall(e, el)
  { if (e->rev == nil)
    { edge r = new_edge(target(e), source(e));
      e->rev = r;
      r->rev = e;
      R.append(r);
    }
  }
}

```

```

extern bool PLANAR(graph&, bool=false);

```

```

void graph::make_planar_map()
{ if (!graph::make_map())

```

```

        error_handler(1,"graph::make_planar_map: graph is not
bidirected");
        if (!PLANAR(*this,true))
            error_handler(1,"graph::make_planar_map: Graph is not planar.");
        compute_faces();
    }

face graph::new_face(GenPtr i)
{ copy_face_entry(i);
  return add_face(i);
}

face graph::new_face()
{ GenPtr i = 0;
  init_face_entry(i);
  return add_face(i);
}

list<edge> graph::adj_edges(face f) const
{ list<edge> result(f->head);
  edge e1 = face_cycle_succ(f->head);
  while (e1!=f->head)
  { result.append(e1);
    e1 = face_cycle_succ(e1);
  }
  return result;
}

list<node> graph::adj_nodes(face f) const
{ list<node> result(source(f->head));
  edge e1 = face_cycle_succ(f->head);
  while (e1!=f->head)
  { result.append(source(e1));
    e1 = face_cycle_succ(e1);
  }
  return result;
}

list<face> graph::adj_faces(node v) const
{ list<face> result;
  edge e;
  forall_out_edges(e,v) result.append(adj_face(e));
  return result;
}

void graph::print_face(face f) const
{ cout << string("F[%2d]",index(f));
  cout << "(";
  print_face_entry(cout,f->data[0]);
  cout << "): ";
  edge e;
  forall_face_edges(e,f)
    cout << string("[%2d]",index(target(e)));
}

```

```

void graph::compute_faces()
{
    del_all_faces();

    FaceOf = new graph_map(this,1,0);

    edge e;
    forall_edges(e,*this)
    { if (e->rev == nil)
        error_handler(1,"graph::compute_faces: no map (reversal edge
missing)");
        access_face(e) = nil;
    }

    forall_edges(e,*this)
    { if (access_face(e) != nil) continue;
        face f = new_face();
        f->head = e;
        edge e1 = e;
        int count = 0;
        do { access_face(e1) = f;
            e1 = face_cycle_succ(e1);
            count++;
        } while (e1 != e);
        f->sz = count;
    }
}

edge graph::split_map_edge(edge e)
{
    /* splits edge e and its reversal by inserting a new node u (node_inf)

        e                      e          rr
    ----->                ----->  ----->
    (v)                    (v)      (u)      (w)
    <-----                <-----  <-----
        r                      er          r
    returns edge rr
    */

    edge r = e->rev;

    if (r == nil)
        error_handler(1,"graph::split_map_edge(e): reversal of edge e
missing.");

    node v = source(e);
    node w = target(e);
    node u = new_node();

    // remove e and r from corresponding in-lists
    w->del_adj_edge(e,1,1);
    v->del_adj_edge(r,1,1);

    // insert e and r in in-list of u
    e->term[1] = u;
    r->term[1] = u;
}

```



```

    u->append_adj_edge(e,1,1);
    u->append_adj_edge(r,1,1);

    // create reverse edges rr and re
    edge rr = graph::new_edge(u,w);
    edge er = graph::new_edge(u,v);

    set_reversal(e,er);
    set_reversal(r,rr);

    access_face(rr) = access_face(e);
    access_face(er) = access_face(r);

    return rr;
}

edge graph::new_map_edge(edge e1, edge e2)
{
    edge e = graph::new_edge(e1,source(e2));
    edge r = graph::new_edge(e2,source(e1));
    set_reversal(e,r);
    return e;
}

edge graph::split_face(edge e1, edge e2)
{
    face f1 = access_face(e1);
    face f2 = access_face(e2);

    if (f1 != f2)
        error_handler(1,"planar_map::new_edge: new edge must lie in a
face.");

    f2 = new_face();

    edge x = graph::new_edge(e1,source(e2));
    edge y = graph::new_edge(e2,source(e1));
    set_reversal(x,y);

    f1->head = x;
    f2->head = y;

    access_face(x) = f1;

    do { access_face(y) = f2;
        y = face_cycle_succ(y);
    } while (y != f2->head);

    return x;
}

list<edge> graph::triangulate_map()
{
    /* G is a planar map. This procedure triangulates all faces of G
    without introducing multiple edges. The algorithm was suggested by
    Christian Uhrig and Torben Hagerup.

    Description:

```

Triangulating a planar graph G , i.e., adding edges to G to obtain a chordal planar graph, in linear time:

- 1) Compute a (combinatorial) embedding of G .
- 2) Step through the vertices of G . For each vertex u , triangulate those faces incident on u that have not already been triangulated. For each vertex u , this consists of the following:
 - a) Mark the neighbours of u . During the processing of u , a vertex will be marked exactly if it is a neighbour of u .
 - b) Process in any order those faces incident on u that have not already been triangulated. For each such face with boundary vertices $u=x_1, \dots, x_n$,
 - I) If $n=3$, do nothing; otherwise
 - II) If x_3 is not marked, add an edge $\{x_1, x_3\}$, mark x_3 and continue triangulating the face with boundary vertices $x_1, x_3, x_4, \dots, x_n$.
 - III) If x_3 is marked, add an edge $\{x_2, x_4\}$ and continue triangulating the face with boundary vertices $x_1, x_2, x_4, x_5, \dots, x_n$.
 - c) Unmark the neighbours of x_1 .

Proof of correctness:

A) All faces are triangulated.
This is rather obvious.

B) There will be no multiple edges.
During the processing of a vertex u , the marks on neighbours of u clearly prevent us from adding a multiple edge with endpoint u . After the processing of u , such an edge is not added because all faces incident on u have been triangulated. This takes care of edges added in step II).

Whenever an edge $\{x_2, x_4\}$ is added in step III), the presence of an edge $\{x_1, x_3\}$ implies, by a topological argument, that x_2 and x_4 are incident on exactly one common face, namely the face currently being processed. Hence we never add another edge $\{x_2, x_4\}$.

*/

```

node v;
edge x;
list<edge> L;

node_array<int> marked(*this,0);

if ( !make_map() )
error_handler(1,"TRIANGULATE_PLANAR_MAP: graph is not a map.");

forall_nodes(v,*this)
{
  list<edge> E1 = adj_edges(v);
  edge e,e1,e2,e3;

```

```

forall(e1,E1) marked[target(e1)]=1;

forall(e,E1)
{
    e1 = e;
    e2 = face_cycle_succ(e1);
    e3 = face_cycle_succ(e2);

    while (target(e3) != v)
        // e1,e2 and e3 are the first three edges in a clockwise
        // traversal of a face incident to v and t(e3) is not equal
        // to v.
        if ( !marked[target(e2)] )
        { // we mark w and add the edge {v,w} inside F, i.e., after
          // dart e1 at v and after dart e3 at w.

            marked[target(e2)] = 1;
            L.append(x = new_edge(e3,source(e1)));
            L.append(e1 = new_edge(e1,source(e3)));
            set_reversal(x,e1);
            e2 = e3;
            e3 = face_cycle_succ(e2);
        }
        else
        { // we add the edge {source(e2),target(e3)} inside F, i.e.,
          // after dart e2 at source(e2) and before dart
          // reversal_of[e3] at target(e3).

            e3 = face_cycle_succ(e3);
            L.append(x = new_edge(e3,source(e2)));
            L.append(e2 = new_edge(e2,source(e3)));
            set_reversal(x,e2);
        }
    //end of while

} //end of stepping through incident faces

node w;
forall_adj_nodes(w,v) marked[w] = 0;

} // end of stepping through nodes

return L;
}

face graph::join_faces(edge x)
{
    edge y = reversal(x);

    if (y == nil)
        error_handler(1,"join_faces: graph must be a map.");

    if (access_face(x) == nil || access_face(y) == nil)
        error_handler(1,"join_faces: no face associated with edges.");

    edge e = face_cycle_succ(y);
    face F1 = adj_face(x);
    face F2 = adj_face(y);

```

```

    if (F1 != F2)
    { edge e = face_cycle_succ(y);
      F1->head = e;
      while ( e != y )
      { access_face(e) = F1;
        e = face_cycle_succ(e);
      }
      clear_face_entry(F2->data[0]);
      del_face(F2);
    }
    else
    { e = face_cycle_succ(e);
      if (e != y) // no isolated edge
        F1->head = e;
      else
      { clear_face_entry(F1->data[0]);
        del_face(F1);
        F1 = F2;
      }
    }

    graph::del_edge(x);
    graph::del_edge(y);

    return F1;
}

void graph::make_bidirected(list<edge>& L)
{ Make_Bidirected(*this,L); }

bool graph::is_bidirected() const
{ edge_array<edge> rev(*this,0);
  return Is_Bidirected(*this,rev);
}

bool graph::is_map() const
{ return Is_Map(*this); }

```

```

/*****
+
+ LEDA 3.5.1
+
+ _g_misc.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****/
#include <LEDA/graph.h>
#include <LEDA/ugraph.h>
#include <LEDA/graph_alg.h>

//-----
// S. Naehrer
//      last modified ( April 1997)
//-----

node_array<int>* num_ptr;

static int source_num(const edge& e) { return (*num_ptr)[source(e)]; }
static int target_num(const edge& e) { return (*num_ptr)[target(e)]; }

bool Is_Simple(const graph& G)
{
    // return true iff G is simple, i.e, has no parallel edges

    list<edge> el= G.all_edges();

    if (el.empty()) return true;

    int n = 0;

    node_array<int> num(G);

    node v;
    forall_nodes(v,G) num[v]= n++;

    num_ptr= &num;
    el.bucket_sort(0,n-1,&source_num);
    el.bucket_sort(0,n-1,&target_num);

    edge e0 = el.pop();
    edge e;
    forall(e,el)
    { if (source(e0) == source(e) && target(e0) == target(e) )
        return false;
    }
}

```

```

        else
            e0 = e;
    }

    return true;
}

list<node> Delete_Loops(graph& G)
{ list<edge> loops;
  list<node> L;
  edge e;
  forall_edges(e,G)
  { node v = source(e);
    node w = target(e);
    if (v == w)
    { L.append(v);
      loops.append(e);
    }
  }
  forall(e,loops) G.del_edge(e);
  return L;
}

list<edge> Make_Simple(graph& G)
{
  list<edge> L;

  //use bucket sort to find and eliminate parallel edges

  list<edge> el = G.all_edges();

  if (el.empty()) return L;

  node_array<int> num(G);
  int n = 0;
  node v;
  forall_nodes(v,G) num[v] = n++;

  num_ptr = &num;

  el.bucket_sort(0,n-1,&source_num);
  el.bucket_sort(0,n-1,&target_num);

  bool deleted = false;
  edge e0 = el.pop();

  edge e;
  forall(e,el)
  { if (source(e0) == source(e) && target(e0) == target(e))
    { G.del_edge(e);
      if (!deleted) L.append(e0);
      deleted = true;
    }
    else
    { deleted = false;
      e0 = e;
    }
  }
}

```

```

    return L;
}

static int edge_ord1(const edge& e) { return index(source(e)); }
static int edge_ord2(const edge& e) { return index(target(e)); }

bool Is_Bidirected(const graph& G, edge_array<edge>& reversal)
{
    // computes for every edge e = (v,w) in G its reversal reversal[e] =
    // (w,v)
    // in G ( nil if not present). Returns true if every edge has a
    // reversal and false otherwise.

    int n = G.max_node_index();
    int count = 0;

    edge e, r;

    forall_edges(e,G) reversal[e] = 0;

    list<edge> El = G.all_edges();
    El.bucket_sort(0,n,&edge_ord2);
    El.bucket_sort(0,n,&edge_ord1);

    list<edge> Ell = G.all_edges();
    Ell.bucket_sort(0,n,&edge_ord1);
    Ell.bucket_sort(0,n,&edge_ord2);

    // merge El and Ell to find corresponding edges

    while (! El.empty() && ! Ell.empty())
    { e = El.head();
      r = Ell.head();

      if (target(r) == source(e))
          if (source(r) == target(e))
              { reversal[e] = r;
                Ell.pop();
                El.pop();
                count++;
              }
          else
              if (index(source(r)) < index(target(e)))
                  Ell.pop();
              else
                  El.pop();

      else
          if (index(target(r)) < index(source(e)))
              Ell.pop();
          else
              El.pop();

    }

    return (count == G.number_of_edges()) ? true : false;
}

```

```
}
```

```
void Make_Bidirected(graph& G, list<edge>& R)
{
    // make graph bi-directed by inserting reversal edges
    // appends new edges to R

    edge_array<edge> rev(G,nil);

    if (Is_Bidirected(G,rev)) return;

    // build list L of edges having no reversals

    list<edge> L;
    edge e;
    forall_edges(e,G)
        if (rev[e] == nil) L.append(e);

    // insert missing reversals
    forall(e,L)
    { edge r = G.new_edge(target(e),source(e));
      R.append(r);
    }
}
```

```
list<edge> Make_Bidirected(graph& G)
{ list<edge> R;
  Make_Bidirected(G,R);
  return R;
}
```

```
static void dfs(node v, int& count1, int& count2, node_array<int>&
dfsnum,
                                node_array<int>&
compnum)
{ dfsnum[v] = ++count1;
  edge e;
  forall_adj_edges(e,v)
  { node w = target(e);
    if (dfsnum[w] == 0)
        dfs(w, count1, count2, dfsnum, compnum);
  }
  compnum[v] = ++count2;
}
```

```
bool Is_Acyclic(const graph& G, list<edge>& back)
{
    //compute dfs and completeion numbers
    node_array<int> dfsnum(G,0);
    node_array<int> compnum(G,0);
    int count1 = 0;
    int count2 = 0;
    node v;
    forall_nodes(v,G)
        if (dfsnum[v] == 0)
            dfs(v, count1, count2, dfsnum, compnum);
}
```



```

    // compute back edges
    back.clear();
    edge e;
    forall_edges(e,G)
    { node v = source(e);
      node w = target(e);
      if (v == w || (dfsnum[v] > dfsnum[w] && compnum[v] < compnum[w]))
        back.append(e);
    }

    return back.empty();
}

bool Is_Acyclic(const graph& G)
{ list<edge> dummy;
  return Is_Acyclic(G,dummy);
}

void Make_Acyclic(graph& G)
{ list<edge> back;
  Is_Acyclic(G,back);
  edge e;
  forall(e,back) G.del_edge(e);
}

static void dfs(const graph& G, node v, node_array<bool>& reached, int&
count)
{ reached[v] = true;
  count++;
  edge e;
  forall_inout_edges(e,v)
  { node w = G.opposite(v,e);
    if ( !reached[w] ) dfs(G,w,reached,count);
  }
}

bool Is_Connected(const graph& G)
{
  node_array <bool>  reached(G, false);
  int count = 0;
  node s = G.first_node();

  if (s != nil)
    dfs(G,s,reached,count);

  return count == G.number_of_nodes();
}

void Make_Connected(graph& G, list<edge>& L)
{
  node_array <bool>  reached(G, false);
  node u = G.first_node();
  int count = 0;

```

```

node v;
forall_nodes(v, G)
  if (!reached[v] )
  { dfs(G,v,reached,count); // explore connected comp with root v
    if (u != v) // link v's comp to the first comp
      L.append(G.new_edge(u, v));
  }
}

list<edge> Make_Connected(graph& G)
{ list<edge> L;
  if (G.number_of_nodes() > 0) Make_Connected(G,L);
  return L;
}

static void make_bicon_dfs(graph& G, node v, int& dfs_count,
                           list<edge>& L,
                           node_array<int>& dfsnum,
                           node_array<int>& lowpt,
                           node_array<node>& parent)
{ node u = nil;

  dfsnum[v] = dfs_count++;
  lowpt[v] = dfsnum[v];

  edge e;
  forall_inout_edges(e,v)
  {
    node w = G.opposite(v,e);

    if (v == w) continue; // ignore loops

    if (u == nil) u = w; // first child

    if ( dfsnum[w] == -1) // w not reached before; e is a tree edge
    {
      parent[w] = v;

      make_bicon_dfs(G, w, dfs_count, L, dfsnum, lowpt, parent);

      if (lowpt[w] == dfsnum[v])
      { // |v| is an articulation point. We now add an edge. If |w| is
the
        // first child and |v| has a parent then we connect |w| and
then
        // |parent[v]|, if |w| is a first child and |v| has no parent
        // we do nothing. If |w| is not the first child then we connect
|w|
        // to the first child. The net effect of all of this is to link
all
        // children of an articulation point to the first child and the
first
        // child to the parent (if it exists)

        if (w == u && parent[v])

```

```

        { L.append(G.new_edge(w, parent[v]));
          //L.append(G.new_edge(parent[v], w)); (if bidirected)
        }

        if (w != u)
        { L.append(G.new_edge(u, w));
          //L.append(G.new_edge(w, u)); (if bidirected)
        }
      }

      lowpt[v] = Min(lowpt[v], lowpt[w]);
    }
    else // non tree edge
      lowpt[v] = Min(lowpt[v], dfsnum[w]);
  }
}

static bool is_bicon_dfs(const graph& G, node v, int& dfs_count,
                        node_array<int>& dfsnum,
                        node_array<int>& lowpt,
                        node_array<node>& parent)
{ node u = nil;
  edge e;

  dfsnum[v] = dfs_count++;
  lowpt[v] = dfsnum[v];

  forall_inout_edges(e,v)
  { node w = G.opposite(v,e);
    if (u == nil) u = w;
    if ( dfsnum[w] == -1 )
    { parent[w] = v;
      if (!is_bicon_dfs(G, w, dfs_count, dfsnum, lowpt, parent))
        return false;
      if (lowpt[w] == dfsnum[v] && (w != u || parent[v])) return
        false;
      lowpt[v] = Min(lowpt[v], lowpt[w]);
    }
    else
      lowpt[v] = Min(lowpt[v], dfsnum[w]);
  }

  return true;
}

bool Is_Biconnected(const graph & G)
{ if (G.empty()) return true;
  if ( ! Is_Connected(G) ) return false;
  node_array<int> lowpt(G);
  node_array<int> dfsnum(G,-1);
  node_array<node> parent(G,nil);
  int dfs_count = 0;
  return is_bicon_dfs(G, G.first_node(), dfs_count, dfsnum, lowpt,
    parent);
}

```

```

void Make_Biconnected(graph& G, list<edge>& L)
{
    if (G.number_of_nodes() == 0) return;

    Make_Connected(G,L);

    node_array<int> lowpt(G);
    node_array<int> dfsnum(G,-1); // dfsnum[v] == -1 <=> v not reached
    node_array<node> parent(G,nil);

    int dfs_count = 0;

    make_bicon_dfs(G, G.first_node(), dfs_count, L, dfsnum, lowpt,
parent);
}

```

```

list<edge> Make_Biconnected(graph & G)
{ list<edge> L;
  Make_Biconnected(G,L);
  return L;
}

```

```

static bool bi_bfs(const graph& G, node s, node_array<int>& side)
{
    list<node> Q;

    Q.append(s);
    side[s] = 0;

    while ( ! Q.empty() )
    { node v = Q.head();
      edge e;
      forall_inout_edges(e,v)
      { node w = G.opposite(v,e);
        if (side[v] == side[w]) return false;
        if (side[w] == -1)
        { Q.append(w);
          side[w] = 1 - side[v];
        }
      }
      Q.pop();
    }

    return true;
}

```

```

bool Is_Bipartite(const graph& G, list<node>& A, list<node>& B)
{
    node_array<int> side(G,-1);
    node v;

```

```

forall_nodes(v,G)
    if (side[v] == -1)
        if (! bi_bfs(G,v,side)) return false;

forall_nodes(v,G)
{ if (side[v] == 0) A.append(v);
  if (side[v] == 1) B.append(v);
}

return true;
}

bool Is_Bipartite(const graph& G)
{ list<node> A,B;
  return Is_Bipartite(G,A,B);
}

int COMPONENTS(const graph&G, node_array<int>&);

int Genus(const graph& G)
{ int n = G.number_of_nodes();
  int m = G.number_of_edges()/2; // G is bidirected

  edge_array<bool> considered(G,false);
  int f = 0;
  edge e;
  forall_edges(e,G)
  { if (G.reversal(e) == nil)
      error_handler(1,"Genus: graph must be a map.");
    if ( !considered[e] )
    { // trace the face to the left of e
      f++;
      edge x = e;
      do { considered[x] = true;
          x = G.face_cycle_succ(x);
        }
        while (x != e);
    }
  }

  node_array<int> cnum(G);
  int c = COMPONENTS(G,cnum)-1;

  return (2-n+m-f+c)/2;
}

/*
void copy_graph(const graph& G, GRAPH<node,edge>& H,
               node_array<node>& v_in_H, edge_array<edge>& e_in_H)
{
  node v;
  forall_nodes(v,G) v_in_H[v] = H.new_node(v);

  edge e;

```

```

    forall_edges(e,G) e_in_H[e] =
        H.new_edge(v_in_H[source(e)],v_in_H[target(e)],e);
}
*/

bool Is_Map(const graph& G)
{ edge_array<edge> rev(G);
  if (!Is_Bidirected(G,rev)) return false;
  edge x;
  forall_edges(x,G)
  { edge y = G.reversal(x);
    if (x != G.reversal(y)) return false;
    if (source(x) != target(y) || source(y) != target(x)) return false;
  }
  return true;
}

bool Is_Planar_Map(const graph& G) { return Is_Map(G) && Genus(G) == 0;
}

bool Is_Planar(const graph& G)
{ graph G1 = G;
  return PLANAR(G1);
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _g_objects.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
+ *****/
#include <LEDA/graph.h>

//-----
// nodes and edges
//
// by S. Naeher (1995)
//-----

node_struct::node_struct(GenPtr inf)
{ data[0] = inf;
  owner = nil;
  id = 0;
  for(int j=0; j<2; j++)
  { first_adj_edge[j] = nil;
    last_adj_edge[j] = nil;
    adj_length[j] = 0;
  }
}

edge_struct::edge_struct(node v, node w, GenPtr i)
{ succ_adj_edge[0] = nil;
  succ_adj_edge[1] = nil;
  pred_adj_edge[0] = nil;
  pred_adj_edge[1] = nil;
  id = 0;
  term[0] = v;
  term[1] = w;
  rev = nil;
  data[0] = i;
}

face_struct::face_struct(GenPtr x)
{ data[0] = x ;
  id = 0;
  owner = nil;
  head = nil;
  sz = 0;
}

```

```

}

void node_struct::append_adj_edge(edge e,int i, int chain_e)
{ // append e to adj_list[i]
  // use succ/pred_adj_edge[chain_e] pointers for chaining of e

  edge last = last_adj_edge[i];

  e->succ_adj_edge[chain_e] = nil;

  if (last == 0) // empty list
  { first_adj_edge[i] = e;
    e->pred_adj_edge[chain_e] = nil;
  }
  else
  { e->pred_adj_edge[chain_e] = last;
    if (source(last) == target(last)) // loop
      last->succ_adj_edge[chain_e] = e;
    else
      last->succ_adj_edge[(this==source(last)) ? 0:1] = e;
  }

  last_adj_edge[i] = e;
  adj_length[i]++;
}

void node_struct::insert_adj_edge(edge e, edge e1, int i, int chain_e,
int dir)
{
  // insert e after (dir==0) or before (dir!=0) e1 into adj_list[i]
  // use succ/pred_adj_edge[chain_e] pointers for chaining

  if (e1 == nil)
  { append_adj_edge(e,i,chain_e);
    return;
  }

  edge e2;          // successor (dir==0) or predecessor (dir!=0) of e1
  int chain_e1; // chaining used for e1

  if (source(e1) == target(e1)) // e1 is a self-loop
    chain_e1 = chain_e;
  else
    chain_e1 = (this == source(e1)) ? 0 : 1;

  if (dir == 0)
  { e2 = e1->succ_adj_edge[chain_e1];
    e->pred_adj_edge[chain_e] = e1;
    e->succ_adj_edge[chain_e] = e2;
    e1->succ_adj_edge[chain_e1] = e;
    if (e2 == nil)
      last_adj_edge[i] = e;
  }
}

```



```

    else
    { if (source(e2) == target(e2)) //loop
      e2->pred_adj_edge[chain_e] = e;
      else
      e2->pred_adj_edge[(this==source(e2)) ? 0:1] = e;
    }
  }
else
{ e2 = e1->pred_adj_edge[chain_e1];
  e->succ_adj_edge[chain_e] = e1;
  e->pred_adj_edge[chain_e] = e2;
  e1->pred_adj_edge[chain_e1] = e;
  if (e2 == nil)
    first_adj_edge[i] = e;
  else
  { if (source(e2) == target(e2)) //loop
    e2->succ_adj_edge[chain_e] = e;
    else
    e2->succ_adj_edge[(this==source(e2)) ? 0:1] = e;
  }
}

adj_length[i]++;
}

void node_struct::del_adj_edge(edge e, int i, int chain_e)
{
  // remove e from adj_list[i]
  // with respect to succ/pred_adj_edge[chain_e] pointers

  edge e_succ = e->succ_adj_edge[chain_e];
  edge e_pred = e->pred_adj_edge[chain_e];

  if (e_succ)
    if (source(e_succ) == target(e_succ)) // loop
      e_succ->pred_adj_edge[chain_e] = e_pred;
    else
      e_succ->pred_adj_edge[(this==source(e_succ)) ? 0:1 ] = e_pred;
  else
    last_adj_edge[i] = e_pred;

  if (e_pred)
    if (source(e_pred) == target(e_pred)) // loop
      e_pred->succ_adj_edge[chain_e] = e_succ;
    else
      e_pred->succ_adj_edge[(this==source(e_pred)) ? 0:1 ] = e_succ;
  else
    first_adj_edge[i] = e_succ;

  adj_length[i]--;
}

void graph_obj_list::clear()
{ obj_list_head = 0;

```

```

    obj_list_tail = 0;
    obj_list_sz = 0;
}

void graph_obj_list::append(graph_object* e)
{
    e->obj_list_succ = 0;

    if (obj_list_sz > 0)
        obj_list_tail->obj_list_succ = e;
    else
        obj_list_head = e;

    e->obj_list_pred = obj_list_tail;
    obj_list_tail = e;

    obj_list_sz++;
}

graph_object* graph_obj_list::pop()
{
    graph_object* e = obj_list_head;
    if (e)
    {
        graph_object* s = e->obj_list_succ;
        obj_list_head = s;
        if (s)
            s->obj_list_pred = 0;
        else
            obj_list_tail = 0;
        obj_list_sz--;
    }
    return e;
}

void graph_obj_list::remove(graph_object* e)
{
    graph_object* s = e->obj_list_succ;
    graph_object* p = e->obj_list_pred;

    if (s)
    {
        e->obj_list_succ = s->obj_list_succ;
        s->obj_list_pred = p;
    }
    else
        obj_list_tail = p;

    if (p)
    {
        e->obj_list_pred = p->obj_list_pred;
        p->obj_list_succ = s;
    }
    else
        obj_list_head = s;

    obj_list_sz--;
}

void graph_obj_list::conc(graph_obj_list& L)
{
    if (L.obj_list_sz == 0) return;

    if (obj_list_sz > 0)

```

```
    obj_list_tail->obj_list_succ = L.obj_list_head;
else
    obj_list_head = L.obj_list_head;
L.obj_list_head->obj_list_pred = obj_list_tail;
obj_list_tail = L.obj_list_tail;
obj_list_sz += L.obj_list_sz;
L.clear();
}
```

```

/*****
*****
+
+ LEDA 3.5.1
+
+ _g_partition.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****
*****/
#include <LEDA/node_partition.h>

void node_partition::init(const graph& G)
{ node v;
  forall_nodes(v,G) ITEM[v] = partition::make_block(v);
}

```

```

/*****
*****
+
+ LEDA 3.5.1
+
+ _g_random.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****
*****/
#include <LEDA/graph.h>
#include <LEDA/ugraph.h>

//-----
// generators for random graphs
//
// S. Naeher
//-----

// we use the global random integer source "rand_int".

void random_graph(graph& G, int n, int m)
{ /* random graph with n nodes and m edges */

    node* V = new node[n];
    int* deg = new int[n];
    int i;

    G.clear();

    for(i=0; i<n; i++)
    { V[i] = G.new_node();
      deg[i] = 0;
    }

    for(i=0; i<m; i++) deg[rand_int(0,n-1)]++;

    for(i=0; i<n; i++)
    { node v = V[i];
      int d = deg[i];
      while (d--) G.new_edge(v,V[rand_int(0,n-1)]);
    }

    delete[] V;
    delete[] deg;
}

void random_ugraph(ugraph& G, int n, int m)

```

```

{ int i;
  node* V = new node[n];

  G.clear();

  for(i=0;i<n;i++) V[i] = G.new_node();

  while (m--) G.new_edge(V[rand_int(0,n-1)],V[rand_int(0,n-1)]);
}

void random_bigraph(graph& G,int n1,int n2,int m,list<node>&
A,list<node>& B)
{
  int d = m/n1;
  int r = m%n1;

  node* AV = new node[n1];
  node* BV = new node[n2];

  A.clear();
  B.clear();
  G.clear();

  for(int a = 0; a < n1; a++) A.append(AV[a] = G.new_node());
  for(int b = 0; b < n2; b++) B.append(BV[b] = G.new_node());

  node v;
  int i;

  forall(v,A)
    for(i=0;i<d;i++)
      G.new_edge(v,BV[rand_int(0,n2-1)]);

  while (r--) G.new_edge(AV[rand_int(0,n1-1)], BV[rand_int(0,n2-1)]);

  delete[] AV;
  delete[] BV;
}

//-----
// random planar graph
//-----

#include <LEDA/sortseq.h>
#include <LEDA/prio.h>
#include <math.h>

#define YNIL seq_item(nil)
#define XNIL pq_item(nil)

#define EPS 0.00001
#define EPS2 0.0000000001

class POINT;

```

```

class SEGMENT;
typedef POINT* point;
typedef SEGMENT* segment;

enum point_type {Intersection=0,Rightend=1,Leftend=2};

class POINT
{
    friend class SEGMENT;

    segment seg;
    int      kind;
    double   x;
    double   y;

public:

    POINT(double a,double b)
    {
        x=a; y=b; seg=0; kind=Intersection;
    }

    LEDA_MEMORY(POINT);

    friend double   get_x(point p)      { return p->x; }
    friend double   get_y(point p)      { return p->y; }
    friend int      get_kind(point p)   { return p->kind; }
    friend segment  get_seg(point p)    { return p->seg; }

    friend bool intersection(segment, segment, point&);
};

static int compare(const point& p1, const point& p2)
{ if (p1==p2) return 0;

    double diffx = get_x(p1) - get_x(p2);
    if (diffx > EPS2 ) return 1;
    if (diffx < -EPS2 ) return -1;

    int      diffk = get_kind(p1)-get_kind(p2);
    if (diffk != 0) return diffk;

    double diffy = get_y(p1) - get_y(p2);
    if (diffy > EPS2 ) return 1;
    if (diffy < -EPS2 ) return -1;

    return 0;
}

class SEGMENT
{
    point startpoint;
    point endpoint;
    double slope;
    double yshift;
    node  left_node;

```

```

int    orient;
int    color;
int    name;

public:

SEGMENT(point, point,int,int);

~SEGMENT() { delete startpoint; delete endpoint; }

LEDA_MEMORY(SEGMENT);

friend point get_startpoint(segment seg)      { return seg->startpoint; }
}
friend point get_endpoint(segment seg)        { return seg->endpoint; }
friend double get_slope(segment seg)          { return seg->slope; }
friend double get_yshift(segment seg)         { return seg->yshift; }
friend node get_left_node(segment seg)        { return seg->
left_node; }
friend void set_left_node(segment seg, node v) { seg->left_node = v; }

friend bool intersection(segment, segment, point&);
};

SEGMENT::SEGMENT(point p1,point p2,int c, int n)
{
    left_node  = nil;
    color      = c;
    name       = n;

    if (compare(p1,p2) < 0)
    { startpoint = p1;
      endpoint   = p2;
      orient     = 0;
    }
    else
    { startpoint = p2;
      endpoint   = p1;
      orient     = 1;
    }

    startpoint->kind = Leftend;
    endpoint->kind   = Rightend;
    startpoint->seg  = this;
    endpoint->seg    = this;

    if (endpoint->x != startpoint->x)
    {
        slope = (endpoint->y - startpoint->y)/(endpoint->x - startpoint-
>x);
        yshift = startpoint->y - slope * startpoint->x;

        startpoint->x -= EPS;
        startpoint->y -= EPS * slope;
        endpoint->x  += EPS;
        endpoint->y  += EPS * slope;
    }
}

```



```

    }
    else //vertical segment
    { startpoint->y -= EPS;
      endpoint->y += EPS;
      slope = 0;
      yshift = 0;
    }
  }

static double x_sweep;
static double y_sweep;

static int compare(const segment& s1, const segment& s2)
{
  double y1 = get_slope(s1)*x_sweep+get_yshift(s1);
  double y2 = get_slope(s2)*x_sweep+get_yshift(s2);

  double diff = y1-y2;
  if (diff > EPS2 ) return 1;
  if (diff < -EPS2 ) return -1;

  if (get_slope(s1) == get_slope(s2))
    return compare(get_x(get_startpoint(s1)),
get_x(get_startpoint(s2)));

  if (y1 <= y_sweep+EPS2)
    return compare(get_slope(s1),get_slope(s2));
  else
    return compare(get_slope(s2),get_slope(s1));
}

static priority_queue<seq_item,point> X_structure;
static sortseq<segment,pq_item> Y_structure;

bool intersection(segment seg1,segment seg2, point& inter)
{
  if (seg1->slope == seg2->slope)
    return false;
  else
  {
    double cx = (seg2->yshift - seg1->yshift) / (seg1->slope - seg2->slope);

    if (cx <= x_sweep) return false;

    if (seg1->startpoint->x > cx || seg2->startpoint->x > cx ||
        seg1->endpoint->x < cx || seg2->endpoint->x < cx ) return false;

    inter = new POINT(cx,seg1->slope * cx + seg1->yshift);
    return true;
  }
}

```

```

inline pq_item Xinsert(seq_item i, point p)
{ return X_structure.insert(i,p); }

inline point Xdelete(pq_item i)
{ point p = X_structure.inf(i);
  X_structure.del_item(i);
  return p;
}

void random_planar_graph(graph& G, node_array<double>& xcoord,
                        node_array<double>& ycoord, int n)
{
  point    p,inter;
  segment  seg, l,lsit,lpred,lsucc,lpredpred;
  pq_item  pqit,pxmin;
  seq_item sitmin,sit,sitpred,sitsucc,sitpredpred;

  int MAX_X = n;
  int MAX_Y = n;

  int N = n;  // number of random segments

  G.clear();

  xcoord.init(G,n,0);
  ycoord.init(G,n,0);

  int count=1;

  //initialization of the X-structure

  for (int i = 0; i < N; i++)
  { //point p = new POINT(rand_int(0,MAX_X/3), rand_int(0,MAX_Y));
    //point q = new POINT(rand_int(2*MAX_X/3,MAX_X),
  rand_int(0,MAX_Y));
    point p = new POINT(rand_int(0,MAX_X), rand_int(0,MAX_Y));
    point q = new POINT(rand_int(0,MAX_X), rand_int(0,MAX_Y));
    seg = new SEGMENT(p,q,0,count++);
    Xinsert(YNIL,get_startpoint(seg));
  }

  x_sweep = -MAXINT;
  y_sweep = -MAXINT;

  while( !X_structure.empty()  && G.number_of_nodes() < n )
  {
    pxmin = X_structure.find_min();
    p = X_structure.inf(pxmin);

    sitmin = X_structure.key(pxmin);

    Xdelete(pxmin);

    if (sitmin == YNIL) //left endpoint

```

```

{
    l = get_seg(p);
    x_sweep = get_x(p);
    y_sweep = get_y(p);

    node w = G.new_node();
    xcoord[w] = x_sweep;
    ycoord[w] = y_sweep;
    set_left_node(l,w);

    sit = Y_structure.insert(l,XNIL);

    Xinsert(sit,get_endpoint(l));

    sitpred = Y_structure.pred(sit);
    sitsucc = Y_structure.succ(sit);

    if (sitpred != YNIL)
    { if ((pqit = Y_structure.inf(sitpred)) != XNIL)
        delete Xdelete(pqit);

        lpred = Y_structure.key(sitpred);

        Y_structure.change_inf(sitpred,XNIL);

        if (intersection(lpred,l,inter))
            Y_structure.change_inf(sitpred,Xinsert(sitpred,inter));
    }

    if (sitsucc != YNIL)
    { lsucc = Y_structure.key(sitsucc);
        if (intersection(lsucc,l,inter))
            Y_structure.change_inf(sit,Xinsert(sit,inter));
    }
}

else if (get_kind(p) == Rightend)
    //right endpoint
    {
        x_sweep = get_x(p);
        y_sweep = get_y(p);

        sit = sitmin;

        sitpred = Y_structure.pred(sit);
        sitsucc = Y_structure.succ(sit);

        segment seg = Y_structure.key(sit);

        Y_structure.del_item(sit);

        delete seg;

        if((sitpred != YNIL)&&(sitsucc != YNIL))
        {
            lpred = Y_structure.key(sitpred);
            lsucc = Y_structure.key(sitsucc);
            if (intersection(lsucc,lpred,inter))
                Y_structure.change_inf(sitpred,Xinsert(sitpred,inter));
        }
    }
}

```

```

    }
else // intersection point p
{
    node w = G.new_node();
    xcoord[w] = get_x(p);
    ycoord[w] = get_y(p);

    /* Let L = list of all lines intersecting in p

        we compute sit      = L.head();
        and      sitpred = L.tail();

        by scanning the Y_structure in both directions
        starting at sitmin;

    */

    /* search for sitpred upwards from sitmin: */
    Y_structure.change_inf(sitmin,XNIL);
    sitpred = Y_structure.succ(sitmin);

    while ((pqit=Y_structure.inf(sitpred)) != XNIL)
    { point q = X_structure.inf(pqit);
      if (compare(p,q) != 0) break;
      X_structure.del_item(pqit);
      Y_structure.change_inf(sitpred,XNIL);
      sitpred = Y_structure.succ(sitpred);
    }

    /* search for sit downwards from sitmin: */
    sit = sitmin;
    seq_item sitl;

    while ((sitl=Y_structure.pred(sit)) != YNIL)
    { pqit = Y_structure.inf(sitl);
      if (pqit == XNIL) break;
      point q = X_structure.inf(pqit);
      if (compare(p,q) != 0) break;
      X_structure.del_item(pqit);
      Y_structure.change_inf(sitl,XNIL);
      sit = sitl;
    }

    // insert edges to p for all segments in sit, ..., sitpred
    // and set left node to w
    lsit = Y_structure.key(sitpred);
    node v = get_left_node(lsit);
    if (v!=nil && w!=nil) G.new_edge(v,w);
    set_left_node(lsit,w);
}

```

into G

```

for(sit1=sit; sit1!=sitpred; sit1 = Y_structure.succ(sit1))
{ lsit = Y_structure.key(sit1);

  v = get_left_node(lsit);
  if (v!=nil && w!=nil) G.new_edge(v,w);
  set_left_node(lsit,w);
}

lsit = Y_structure.key(sit);
lpred=Y_structure.key(sitpred);
sitpredpred = Y_structure.pred(sit);
sitsucc=Y_structure.succ(sitpred);

if (sitpredpred != YNIL)
{
  lpredpred=Y_structure.key(sitpredpred);

  if ((pqit = Y_structure.inf(sitpredpred)) != XNIL)
    delete Xdelete(pqit);

  Y_structure.change_inf(sitpredpred,XNIL);

  if (intersection(lpred,lpredpred,inter))
    Y_structure.change_inf(sitpredpred,
                          Xinsert(sitpredpred,inter));
}

if (sitsucc != YNIL)
{
  lsucc=Y_structure.key(sitsucc);

  if ((pqit = Y_structure.inf(sitpred)) != XNIL)
    delete Xdelete(pqit);

  Y_structure.change_inf(sitpred,XNIL);

  if (intersection(lsucc,lsit,inter))
    Y_structure.change_inf(sit,Xinsert(sit,inter));
}

// reverse the subsequence sit, ... ,sitpred in the Y_structure

x_sweep = get_x(p);
y_sweep = get_y(p);

Y_structure.reverse_items(sit,sitpred);

delete p;

} // intersection

}

pq_item xit;
forall_items(xit,X_structure)
{ point      p = X_structure.inf(xit);

```

```

        seq_item sit = X_structure.key(xit);
        if (get_kind(p) == Leftend)      delete get_seg(p);
        if (get_kind(p) == Rightend && sit) delete Y_structure.key(sit);
        if (get_kind(p) == Intersection) delete p;
    }

    X_structure.clear();
    Y_structure.clear();

    Make_Connected(G);

    // normalize x and y coordinates
    node v;
    forall_nodes(v,G)
    { xcoord[v] /= x_sweep;
      ycoord[v] /= n;
    }
}

void random_planar_graph(graph& G, int n)
{ node_array<double> xcoord;
  node_array<double> ycoord;
  random_planar_graph(G,xcoord,ycoord,n);
  //random_planar_graph(G,n,n);
  //Make_Connected(G);
}

void maximal_planar_map(graph& G, int n)
{
    G.clear();

    if (n <= 0 ) return;

    node a = G.new_node();
    n--;

    if (n == 0) return;

    node b = G.new_node();
    n--;

    edge* E = new edge[6*n];

    E[0] = G.new_edge(a,b);
    E[1] = G.new_edge(b,a);
    G.set_reversal(E[0],E[1]);

    int m = 2;

    while (n--)
    { edge e = E[rand_int(0,m-1)];
      node v = G.new_node();
      while (target(e) != v)
      { edge x = G.new_edge(v,source(e));
        edge y = G.new_edge(e,v,after);
      }
    }
}

```

```

        E[m++] = x;
        E[m++] = y;
        G.set_reversal(x,y);
        e = G.face_cycle_succ(e);
    }
}

delete[] E;
}

void maximal_planar_graph(graph& G, int n)
{
    maximal_planar_map(G,n);
    list<edge> E;

    edge_array<bool> marked(G,false);
    edge e;
    forall_edges(e,G)
    { if (!marked[e]) E.append(e);
      marked[e] = true;
      marked[G.reversal(e)] = true;
    }

    forall(e,E) G.del_edge(e);
}

void random_planar_map(graph& G, int n, int m)
{
    maximal_planar_map(G,n);

    list<edge> E;

    edge_array<bool> marked(G,false);
    edge e;
    forall_edges(e,G)
    { if (!marked[e]) E.append(e);
      marked[e] = true;
      marked[G.reversal(e)] = true;
    }

    E.permute();

    while (E.length() > m)
    { edge e = E.pop();
      edge r = G.reversal(e);
      G.del_edge(e);
      G.del_edge(r);
    }
}

void random_planar_graph(graph& G, int n, int m)
{
    random_planar_map(G,n,m);
    list<edge> E;

```

```

edge_array<bool> marked(G, false);
edge e;
forall_edges(e, G)
{ if (!marked[e]) E.append(e);
  marked[e] = true;
  marked[G.reversal(e)] = true;
}

forall(e, E) G.del_edge(e);
}

```



```

/*****
+
+ LEDA 3.5.1
+
+ _g_sort.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****/
#include <LEDA/graph.h>

//-----
// sorting
//
// by S. Naehrer (1995)
//-----

static const graph_map* NA;

static graph* GGG;

/*
static int array_cmp_nodes(const node& x, const node& y)
{ return NA->cmp_entry(NA->array_read(x),NA->array_read(y)); }

static int array_cmp_edges(const edge& x, const edge& y)
{ return NA->cmp_entry(NA->array_read(x),NA->array_read(y)); }
*/

static int int_array_cmp_nodes(const node& x, const node& y)
{ return LEDA_COMPARE(int,NA->array_read(x),NA->array_read(y)); }

static int int_array_cmp_edges(const edge& x, const edge& y)
{ return LEDA_COMPARE(int,NA->array_read(x),NA->array_read(y)); }

static int float_array_cmp_nodes(const node& x, const node& y)
{ return LEDA_COMPARE(float,NA->array_read(x),NA->array_read(y)); }

static int float_array_cmp_edges(const edge& x, const edge& y)
{ return LEDA_COMPARE(float,NA->array_read(x),NA->array_read(y)); }

static int double_array_cmp_nodes(const node& x, const node& y)
{ return LEDA_COMPARE(double,NA->array_read(x),NA->array_read(y)); }

static int double_array_cmp_edges(const edge& x, const edge& y)
{ return LEDA_COMPARE(double,NA->array_read(x),NA->array_read(y)); }

```

```

static int graph_cmp_nodes(const node& x, const node& y)
{ return GGG->cmp_node_entry(x,y); }

static int graph_cmp_edges(const edge& x, const edge& y)
{ return GGG->cmp_edge_entry(x,y); }

void graph::sort_nodes(const list<node>& vl)
{
    if (vl.length() != number_of_nodes())
        error_handler(1,"graph::sort_nodes(list<node>): illegal node
list");

    v_list.clear();

    node v;
    forall(v,vl)
    { if (v->owner != this)
        error_handler(1,"graph::sort_nodes(list<node>): illegal node
list");
        v_list.append(v);
    }
}

void graph::sort_nodes(int (*f)(const node&, const node&))
{ list<node> vl = all_nodes();
  vl.sort(f);
  sort_nodes(vl);
}

void graph::sort_edges(const list<edge>& el)
{
    node v;
    edge e;

    if (el.length() != number_of_edges())
        error_handler(1,"graph::sort_edges(list<edge>): illegal edge
list");

    // clear all adjacency lists
    forall_nodes(v,*this)
    { for(int i=0; i<2; i++)
        { v->first_adj_edge[i] = 0;
          v->last_adj_edge[i] = 0;
          v->adj_length[i] = 0;
        }
    }

    e_list.clear();

    forall(e,el)
    {
        if (e->term[0]->owner != this)
            error_handler(1,"graph::sort_edges(list<edge>): edge not in
graph");
    }
}

```

```

    e_list.append(e);

    source(e)->append_adj_edge(e,0,0);
    if (undirected)
        target(e)->append_adj_edge(e,0,1);
    else
        target(e)->append_adj_edge(e,1,1);
}

}

void graph::sort_edges(int (*f)(const edge&, const edge&))
{ list<edge> el = all_edges();
  el.sort(f);
  sort_edges(el);
}

void graph::sort_nodes(const graph_map& A)
{ NA = &A;
  switch (A.elem_type_id()) {

    case INT_TYPE_ID: sort_nodes(int_array_cmp_nodes);
                      break;
    case FLOAT_TYPE_ID:
                      sort_nodes(float_array_cmp_nodes);
                      break;
    case DOUBLE_TYPE_ID:
                      sort_nodes(double_array_cmp_nodes);
                      break;

    default:
      error_handler(1,"G.sort_nodes(node_array<T>): T must be
numerical.");
  }
}

void graph::sort_edges(const graph_map& A)
{ NA = &A;
  switch (A.elem_type_id()) {

    case INT_TYPE_ID: sort_edges(int_array_cmp_edges);
                      break;
    case FLOAT_TYPE_ID:
                      sort_edges(float_array_cmp_edges);
                      break;
    case DOUBLE_TYPE_ID:
                      sort_edges(double_array_cmp_edges);
                      break;

    default:
      error_handler(1,"G.sort_edges(node_array<T>): T must be
numerical.");
  }
}

void graph::sort_nodes()
{ GGG = this;
  sort_nodes(graph_cmp_nodes);
}

```

```

void graph::sort_edges()
{ GGG = this;
  sort_edges(graph_cmp_edges);
}

// bucket sort

static int array_ord_node(const node& x)
{ return LEDA_ACCESS(int,NA->array_read(x)); }

static int array_ord_edge(const edge& x)
{ return LEDA_ACCESS(int,NA->array_read(x)); }

void graph::bucket_sort_nodes(int l, int h, int (*ord)(const node&))
{ list<node> vl = all_nodes();
  vl.bucket_sort(l,h,ord);
  sort_nodes(vl);
}

void graph::bucket_sort_edges(int l, int h, int (*ord)(const edge&))
{ list<edge> el = all_edges();
  el.bucket_sort(l,h,ord);
  sort_edges(el);
}

void graph::bucket_sort_nodes(int (*ord)(const node&))
{ list<node> vl = all_nodes();
  vl.bucket_sort(ord);
  sort_nodes(vl);
}

void graph::bucket_sort_edges(int (*ord)(const edge&))
{ list<edge> el = all_edges();
  el.bucket_sort(ord);
  sort_edges(el);
}

void graph::bucket_sort_nodes(const graph_map& A)
{ NA = &A;
  switch (A.elem_type_id()) {

    case INT_TYPE_ID: bucket_sort_nodes(array_ord_node);
                      break;

    default:
      error_handler(1,"G.bucket_sort_nodes(node_array<T>): T must be
integer.");
  }
}

void graph::bucket_sort_edges(const graph_map& A)
{ NA = &A;
  switch (A.elem_type_id()) {

    case INT_TYPE_ID: bucket_sort_edges(array_ord_edge);

```

```
                break;
    default:
        error_handler(1, "G.bucket_sort_edges(edge_array<T>): T must be
integer.");
    }
}
```

```

/*****
*****
+
+ LEDA 3.5.1
+
+ _gml_graph.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****
*****/
//----- //
// class gml_graph, parser for graphs in GML format //
// //
// by David Alberts (1997) //
//----- //

#include<LEDA/gml_graph.h>

void gml_graph::init_rules()
{
    // graph rule
    append("graph");
    add_rule(new_graph,gml_list);

    // directed graph (default) or not?
    append("directed");
    add_rule(directed,gml_int);
    goback();

    // type of node parameter (optional)
    append("nodeType");
    add_rule(nodeType,gml_string);
    goback();

    // type of edge parameter (optional)
    append("edgeType");
    add_rule(edgeType,gml_string);
    goback();

    // new node
    append("node");
    add_rule(new_node,gml_list);

    // node index
    append("id");
    add_rule(node_index,gml_int);
    goback();

    // node parameter
    append("parameter");
    add_rule(node_param,gml_string);

```

```

goback();

goback();

// new edge
append("edge");
add_rule(new_edge,gml_list);

// edge source
append("source");
add_rule(edge_source,gml_int);
goback();

// edge target
append("target");
add_rule(edge_target,gml_int);
goback();

// edge parameter
append("parameter");
add_rule(edge_param,gml_string);

reset_path();
}

void gml_graph::add_graph_rule(gml_graph_rule f, gml_value_type t, char*
key)
{
    if(key)
    {
        reset_path();
        append("graph");
        append(key);
    }

    graph_rules[next_rule] = f;
    add_rule(next_rule,t);
    next_rule++;

    if(key) reset_path();
}

void gml_graph::add_node_rule(gml_node_rule f, gml_value_type t, char*
key)
{
    if(key)
    {
        reset_path();
        append("graph");
        append("node");
        append(key);
    }

    node_rules[next_rule] = f;
    add_rule(next_rule,t);
    next_rule++;

    if(key) reset_path();
}

```

```

void gml_graph::add_edge_rule(gml_edge_rule f, gml_value_type t, char*
key)
{
    if(key)
    {
        reset_path();
        append("graph");
        append("edge");
        append(key);
    }

    edge_rules[next_rule] = f;
    add_rule(next_rule,t);
    next_rule++;

    if(key) reset_path();
}

bool gml_graph::interpret(gml_rule r, const gml_object* gobj)
{
    bool ok = true;

    switch(r)
    {
        case new_graph:
        {
            ok = graph_intro(gobj);
            break;
        }
        case directed:
        {
            if(gobj->get_int()) the_graph->make_directed();
            else the_graph->make_undirected();
            break;
        }
        case nodeType:
        {
            right_node_type = !strcmp(the_graph->node_type(),gobj-
>get_string());
            break;
        }
        case edgeType:
        {
            right_edge_type = !strcmp(the_graph->edge_type(),gobj-
>get_string());
            break;
        }
        case new_node:
        {
            current_node = the_graph->new_node();
            has_id = false;
            gml_node_rule f;
            forall(f,new_node_rules) ok = ok &&
(*f)(gobj,the_graph,current_node);
            break;
        }
        case node_index:
        {
            (*(node_by_id))[gobj->get_int()] = current_node;
            has_id = true;
        }
    }
}

```



```

        break;
    }
    case node_param:
    {
        the_graph->set_node_entry(current_node, string(gobj-
>get_string()));
        break;
    }
    case new_edge:
    {
        edge e = the_graph->new_edge(dummy1, dummy2);
        (*(edge_s))[e] = -1;
        (*(edge_t))[e] = -1;
        current_edge = e;
        gml_edge_rule f;
        forall(f, new_edge_rules) ok = ok &&
(*f)(gobj, the_graph, current_edge);
        break;
    }
    case edge_source:
    {
        (*(edge_s))[current_edge] = gobj->get_int();
        break;
    }
    case edge_target:
    {
        (*(edge_t))[current_edge] = gobj->get_int();
        break;
    }
    case edge_param:
    {
        the_graph->set_edge_entry(current_edge, string(gobj-
>get_string()));
        break;
    }
    default:
    {
        if(node_rules.defined(r))
        {
            ok = (*(node_rules[r]))(gobj, the_graph, current_node);
            break;
        }
        if(edge_rules.defined(r))
        {
            ok = (*(edge_rules[r]))(gobj, the_graph, current_edge);
            break;
        }
        if(graph_rules.defined(r))
        {
            ok = (*(graph_rules[r]))(gobj, the_graph);
            break;
        }
        break;
    }
}

return ok;
}

bool gml_graph::list_end(gml_rule r, const gml_object* gobj)
{

```

```

bool ok = true;

switch(r)
{
    case new_graph:
    {
        ok = graph_end(gobj);
        break;
    }
    case new_node:
    {
        if(!has_id)
        {
            print_error(*gobj,"missing node id");
            ok = false;
        }

        gml_node_rule f;
        forall(f,node_done_rules) ok = ok &&
(*f)(gobj,the_graph,current_node);

        current_node = nil;

        break;
    }
    case new_edge:
    {
        ok = edge_end(gobj);
        break;
    }
    default:
    {
        break;
    }
}

return ok;
}

bool gml_graph::graph_intro(const gml_object* gobj)
{
    node_by_id = new map<int,node>;
    edge_s = new map<edge,int>;
    edge_t = new map<edge,int>;

    the_graph->clear();
    dummy1 = the_graph->new_node();
    dummy2 = the_graph->new_node();

    right_node_type = true;
    right_edge_type = true;

    // call new graph rules
    bool ok = true;
    gml_graph_rule f;
    forall(f,new_graph_rules) ok = ok && (*f)(gobj,the_graph);

    return ok;
}

```

```

bool gml_graph::graph_end(const gml_object* gobj)
{
    bool ok = true;

    if(!right_node_type || !right_edge_type)
    {
        if(!right_node_type)
            print_error(*gobj, "wrong node type");
        if(!right_edge_type)
            print_error(*gobj, "wrong edge type");
        ok = false;
    }

    edge e;
    node s,t;
    // settle edges
    forall_edges(e,*the_graph)
    {
        s = t = 0;
        if((*edge_s)[e] != -1)
            s = (*(node_by_id))[(*edge_s)[e]];
        if((*edge_t)[e] != -1)
            t = (*(node_by_id))[(*edge_t)[e]];
        if(s && t) the_graph->move_edge(e,s,t);
    }

    the_graph->del_node(dummy1);
    the_graph->del_node(dummy2);

    // call graph done rules
    gml_graph_rule f;
    forall(f,graph_done_rules) ok = ok && (*f)(gobj,the_graph);

    if(node_by_id)
    {
        delete node_by_id;
        delete edge_s;
        delete edge_t;

        node_by_id = 0;
        edge_s = 0;
        edge_t = 0;
    }

    return ok;
}

bool gml_graph::edge_end(const gml_object* gobj)
{
    bool ok = true;

    if((*edge_s)[current_edge] == -1)
    {
        print_error(*gobj, "edge without source");
        ok = false;
    }
    if((*edge_t)[current_edge] == -1)
    {
        print_error(*gobj, "edge without target");
    }
}

```

```
    ok = false;
}

gml_edge_rule f;
forall(f, edge_done_rules) ok = ok &&
(*f)(gobj, the_graph, current_edge);

current_edge = nil;

return ok;
}
```

```

/*****
+
+ LEDA 3.5.1
+
+ _graph.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****/
#include <LEDA/graph.h>

//-----
// basic graph operations
//
// by S. Naeher (1995,1996)
//-----

graph::graph()
{ int sz1 = LEDA::node_data_slots;
  int sz2 = LEDA::edge_data_slots;
  max_n_index = -1;
  max_e_index = -1;
  max_f_index = -1;
  parent = 0;
  undirected = false;
  data_sz[0] = sz1;
  data_sz[1] = sz2;
  data_sz[2] = 0;
  while (sz1) free_data[0].push(sz1--);
  while (sz2) free_data[1].push(sz2--);
  node_data_map = new graph_map(this,0);
  edge_data_map = new graph_map(this,1);
  adj_iterator = new graph_map(this,0,0);
  FaceOf = 0;
}

graph::graph(int sz1, int sz2)
{ max_n_index = -1;
  max_e_index = -1;
  max_f_index = -1;
  parent = 0;
  undirected = false;
  data_sz[0] = sz1;
  data_sz[1] = sz2;
  data_sz[2] = 0;
  while (sz1) free_data[0].push(sz1--);
  while (sz2) free_data[1].push(sz2--);
  node_data_map = new graph_map(this,0);
}

```

```

    edge_data_map = new graph_map(this,1);
    adj_iterator = new graph_map(this,0,0);
    FaceOf = 0;
}

void graph::copy_all_entries() const
{
    node v;
    forall_nodes(v,*this) copy_node_entry(v->data[0]);
    edge e;
    forall_edges(e,*this) copy_edge_entry(e->data[0]);
    // hidden edges
    for(e = (edge)h_list.head(); e; e = (edge)h_list.succ(e))
        copy_edge_entry(e->data[0]);
}

void graph::clear_all_entries() const
{
    node v;
    forall_nodes(v,*this) clear_node_entry(v->data[0]);
    edge e;
    forall_edges(e,*this) clear_edge_entry(e->data[0]);
    // hidden edges
    for(e = (edge)h_list.head(); e; e = (edge)h_list.succ(e))
        clear_edge_entry(e->data[0]);
}

void graph::copy_graph(const graph& G)
{
    int n = G.number_of_nodes();
    //int m = G.number_of_edges();

    for(int k = 0; k < 3; k++)
    {
        data_sz[k] = G.data_sz[k];
        for(int i=1; i<=data_sz[k]; i++) free_data[k].append(i);
    }

    max_n_index = -1;
    max_e_index = -1;
    max_f_index = -1;

    e_list.clear();

    FaceOf = 0;

    parent = 0;

    if (n == 0) return;

    node* node_vec = new node[G.max_n_index+1];
    edge* edge_vec = new edge[G.max_e_index+1];

    if (node_vec == 0 || edge_vec == 0)
        error_handler(1," copy_graph: out of memory");

    // allocate a single block of memory for all nodes
    // memory_allocate_block(sizeof(node_struct),n);

    node v;

```

```

forall_nodes(v,G)
    node_vec[index(v)] = new_node(v->data[0]);

// allocate a single block of memory for all edges
// memory_allocate_block(sizeof(edge_struct),m);

bool loops_deleted = false;

forall_nodes(v,G)
{
    node a = node_vec[index(v)];
    edge e;
    forall_adj_edges(e,v)
    {
        if ( undirected && v == target(e) ) //incoming edge
        {
            if (v == source(e)) // loop
                loops_deleted = true;
        }
        else
        {
            node b = node_vec[index(target(e))];
            edge_vec[index(e)] = new_edge(a,b,e->data[0]);
        }
    }
}

// update reversal information
edge e;
forall_edges(e,G)
    if (e->rev)
        edge_vec[index(e)]->rev = edge_vec[index(e->rev)];

// copy faces (if existing)
face f;
forall_faces(f,G)
{
    face f1 = new_face(f->data[0]);
    f1->head = edge_vec[index(f->head)];
}

delete[] node_vec;
delete[] edge_vec;

if ( loops_deleted )
    error_handler(0,"selfloops deleted in ugraph copy constructor");
}

graph::graph(const graph& G)
{
    undirected = G.undirected;
    copy_graph(G);
    node_data_map = new graph_map(this,0);
    edge_data_map = new graph_map(this,1);
    adj_iterator = new graph_map(this,0,0);
}

graph& graph::operator=(const graph& G)
{
    if (&G != this)
    {
        graph::clear();
        undirected = G.undirected;
        copy_graph(G);
    }
}

```

```

    }
    return *this;
}

void graph::join(graph& G)
{ // moves all objects from G to this graph and clears G

    if (G.undirected != undirected)
        error_handler(1,"graph::join(G): cannot merge directed and
undirected graphs.");

    for(int d=0; d<3; d++) {
        if (G.data_sz[d] != data_sz[d])
            error_handler(1,"graph::join(G): cannot merge graphs with
different data sizes.");
    }

    node v;
    edge e;
    face f;

    int i = max_n_index;
    forall_nodes(v,G) { v->id = ++i; v->owner = this; }
    max_n_index = i;

    int j = max_e_index;
    forall_edges(e,G) e->id = ++j;
    max_e_index = j;

    int k = max_f_index;
    forall_faces(f,G) f->id = ++k;
    max_f_index = k;

    v_list.conc(G.v_list);
    e_list.conc(G.e_list);
    f_list.conc(G.f_list);

    G.max_n_index = -1;
    G.max_e_index = -1;
    G.max_f_index = -1;
}

```

```

// subgraph constructors (do not work for undirected graphs)
/*
graph::graph(graph& G, const list<node>& nl, const list<edge>& el)
{ // construct subgraph (nl,el) of graph G

    parent = &G;
    node v,w;
    edge e;

    node* N = new node[G.max_n_index+1];

```



```

forall(v,nl)
{ if (graph_of(v) != parent)
  error_handler(1,"graph: illegal node in subgraph constructor");
  N[index(v)] = new_node((GenPtr)v);
}

forall(e,el)
{ v = source(e);
  w = target(e);
  if ( graph_of(e) != parent || N[index(v)]==0 || N[index(w)]==0 )
    error_handler(1,"graph: illegal edge in subgraph constructor");
  new_edge(N[index(v)],N[index(w)],(GenPtr)e);
}

undirected = G.undirected;

delete[] N;
}

graph::graph(graph& G, const list<edge>& el)
{ // construct subgraph of graph G with edge set el

  node v,w;
  edge e;
  node* N = new node[G.max_n_index+1];

  forall_nodes(v,G) N[index(v)] = 0;

  parent = &G;

  forall(e,el)
  { v = source(e);
    w = target(e);
    if (N[index(v)] == 0) N[index(v)] = new_node((GenPtr)v);
    if (N[index(w)] == 0) N[index(w)] = new_node((GenPtr)w);
    if ( graph_of(e) != parent )
      error_handler(1,"graph: illegal edge in subgraph constructor");
    new_edge(N[index(v)],N[index(w)],(GenPtr)e);
  }

  undirected = G.undirected;

  delete[] N;

}

*/

//-----
//
// destruction
//-----

void graph::del_all_nodes() { clear(); }

void graph::del_all_edges()
{

```

```

edge e;

e = (edge)e_list.head();
while (e)
{ edge next = (edge)e_list.succ(e);
  dealloc_edge(e);
  e = next;
}

e = (edge)h_list.head();
while (e)
{ edge next = (edge)h_list.succ(e);
  dealloc_edge(e);
  e = next;
}

e = (edge)e_free.head();
while (e)
{ edge next = (edge)e_free.succ(e);
  dealloc_edge(e);
  e = next;
}

e_list.clear();
h_list.clear();
e_free.clear();

max_e_index = -1;

node v;
forall_nodes(v,*this)
  for(int i=0; i<2; i++)
  { v->first_adj_edge[i] = nil;
    v->last_adj_edge[i] = nil;
    v->adj_length[i] = 0;
  }
}

void graph::del_all_faces()
{
  face f = (face)f_list.head();
  while (f)
  { face next = (face)f_list.succ(f);
    dealloc_face(f);
    f = next;
  }

  f = (face)f_free.head();
  while (f)
  { face next = (face)f_free.succ(f);
    dealloc_face(f);
    f = next;
  }

  f_free.clear();
  f_list.clear();

  if (FaceOf)
  { delete FaceOf;
    FaceOf = 0;
  }
}

```

```

    }

    max_f_index = -1;
}

void graph::clear()
{
    pre_clear_handler();

    for(int k=0; k<3; k++)
    { graph_map* m;
      forall(m,map_list[k])
        if (m->g_index > 0) m->clear_table();
    }

    del_all_faces();
    del_all_edges();

    node v = (node)v_list.head();
    while (v)
    { node next = (node)v_list.succ(v);
      dealloc_node(v);
      v = next;
    }

    v = (node)v_free.head();
    while (v)
    { node next = (node)v_free.succ(v);
      dealloc_node(v);
      v = next;
    }

    v_list.clear();
    v_free.clear();

    max_n_index = -1;

    post_clear_handler();
}

```

```

graph::~~graph()
{ clear();
  for(int k=0; k<3; k++)
  { graph_map* m;
    forall(m,map_list[k]) m->g = 0;
  }
  delete adj_iterator;
  delete node_data_map;
  delete edge_data_map;
}

```

```

//-----
//
// accessing node and edge lists
//-----

```

```

const list<node>& graph::all_nodes() const
{ ((list<node>&)v_list_tmp).Clear();
  node v;
  forall_nodes(v,*this)
    ((list<node>&)v_list_tmp).append(v);
  return v_list_tmp;
}

const list<edge>& graph::all_edges() const
{ ((list<edge>&)e_list_tmp).Clear();
  edge e;
  forall_edges(e,*this)
    ((list<edge>&)e_list_tmp).append(e);
  return e_list_tmp;
}

const list<face>& graph::all_faces() const
{ ((list<face>&)f_list_tmp).Clear();
  face f;
  forall_faces(f,*this)
    ((list<face>&)f_list_tmp).append(f);
  return f_list_tmp;
}

list<edge> graph::out_edges(node v) const
{ list<edge> result;
  edge e;
  forall_out_edges(e,v) result.append(e);
  return result;
}

list<edge> graph::in_edges(node v) const
{ list<edge> result;
  edge e;
  forall_in_edges(e,v) result.append(e);
  return result;
}

list<edge> graph::adj_edges(node v) const
{ list<edge> result;
  edge e;
  forall_adj_edges(e,v) result.append(e);
  return result;
}

list<node> graph::adj_nodes(node v) const
{ list<node> result;
  edge e;
  forall_adj_edges(e,v) result.append(opposite(v,e));
  return result;
}

```

```

//-----
//
// update operations
//-----

```

```

list<edge> graph::insert_reverse_edges()
{ list<edge> L;
  edge e = first_edge();

  if (e != nil)
  { L.append(new_edge(target(e),source(e),e->data[0]));
    copy_edge_entry(e->data[0]);
    e = succ_edge(e);
  }

  edge stop = last_edge();

  while (e != stop)
  { L.append(new_edge(target(e),source(e),e->data[0]));
    copy_edge_entry(e->data[0]);
    e = succ_edge(e);
  }

  return L;
}

```

```

face graph::add_face(GenPtr inf)
{ face f;
  if ( f_free.empty() )
  { f = (face)std_memory.allocate_bytes(face_bytes());
    new (f) face_struct(inf);
    f->owner = this;
    f->id = ++max_f_index;
  }
  else
  { f = (face)f_free.pop();
    f->data[0] = inf;
  }

  f_list.append(f);

  graph_map* m;
  forall(m,map_list[2]) m->re_init_entry(f);

  return f;
}

```

```

void graph::dealloc_face(face f)
{ std_memory.deallocate_bytes(f,face_bytes()); }

```

```

void graph::del_face(face f)
{ f_list.remove(f);
  f_free.append(f);
  graph_map* m;
  forall(m,map_list[2])
  { int i = m->g_index;
    if (i > 0) m->clear_entry(f->data[i]);
  }
}

```

```

    }
}

node graph::add_node(GenPtr inf)
{ node v;
  if ( v_free.empty() )
    { v = (node)std_memory.allocate_bytes(node_bytes());
      new (v) node_struct(inf);
      v->owner = this;
      v->id = ++max_n_index;
      v->succ_link = nil;
    }
  else
    { v = (node)v_free.pop();
      v->data[0] = inf;
    }

  v_list.append(v);

  graph_map* m;
  forall(m, map_list[0]) m->re_init_entry(v);

  return v;
}

void graph::dealloc_node(node v)
{ std_memory.deallocate_bytes(v, node_bytes()); }

node graph::new_node()
{ GenPtr x = 0;
  pre_new_node_handler();
  init_node_entry(x);
  node v = add_node(x);
  post_new_node_handler(v);
  return v;
}

node graph::new_node(GenPtr i)
{ pre_new_node_handler();
  node v = add_node(i);
  post_new_node_handler(v);
  return v;
}

void graph::del_node(node v)
{
  if (v->owner != this)
    error_handler(4, "del_node(v): v is not in G");

  // delete adjacent edges

  edge e;
  while ((e=v->first_adj_edge[0]) != nil) del_edge(e);
}

```

```

    if (!undirected)
        while ((e=v->first_adj_edge[1]) != nil) del_edge(e);

    pre_del_node_handler(v);

    if (parent==0) clear_node_entry(v->data[0]);

    v_list.remove(v);
    v_free.append(v);

    graph_map* m;
    forall(m,map_list[0])
    { int i = m->g_index;
      if (i > 0) m->clear_entry(v->data[i]);
    }

    post_del_node_handler();
}

node graph::merge_nodes(node v1, node v2)
{
    if (undirected)
        error_handler(1,"merge_nodes not implemented for undirected
graphs.");

    for(int i=0; i<2; i++)
    {
        if (v1->last_adj_edge[i])
            v1->last_adj_edge[i]->succ_adj_edge[i] = v2->first_adj_edge[i];
        else
            v1->first_adj_edge[i] = v2->first_adj_edge[i];

        if (v2->first_adj_edge[i])
        { v2->first_adj_edge[i]->pred_adj_edge[i] = v1->last_adj_edge[i];
          v1->last_adj_edge[i] = v2->last_adj_edge[i];
        }

        v1->adj_length[i] += v2->adj_length[i];

        v2->adj_length[i] = 0;
        v2->first_adj_edge[i] = 0;
        v2->last_adj_edge[i] = 0;
    }

    del_node(v2);
    return v1;
}

edge graph::add_edge(node v, node w, GenPtr inf)
{ edge e;

    if (v->owner != this)
        error_handler(6, "new_edge(v,w): v not in graph");

```

```

    if (w->owner != this)
        error_handler(6, "new_edge(v,w): w not in graph");

    if ( e_free.empty() )
    { e = (edge)std_memory.allocate_bytes(edge_bytes());
      new (e) edge_struct(v,w,inf);
      e->id = ++max_e_index;
    }
    else
    { e = (edge)e_free.pop();
      e->data[0] = inf;
      e->term[0] = v;
      e->term[1] = w;
      e->rev = nil;
      e->succ_adj_edge[0] = nil;
      e->succ_adj_edge[1] = nil;
      e->pred_adj_edge[0] = nil;
      e->pred_adj_edge[1] = nil;
    }

    e_list.append(e);

    graph_map* m;
    forall(m,map_list[1]) m->re_init_entry(e);

    return e;
}

void graph::dealloc_edge(edge e)
{ std_memory.deallocate_bytes(e,edge_bytes()); }

void graph::del_adj_edge(edge e, node v, node w)
{ if (undirected)
    { v->del_adj_edge(e,0,0);
      w->del_adj_edge(e,0,1);
    }
    else
    { v->del_adj_edge(e,0,0);
      w->del_adj_edge(e,1,1);
    }
}

void graph::ins_adj_edge(edge e, node v, edge e1, node w, edge e2,int
d1,int d2)
{
    // insert edge e
    // after(if d1=0)/before(if d1=1) e1 to adj_list of v
    // after(if d2=0)/before(if d2=1) e2 to in_list (adj_list) of w
    // (most general form of new_edge)

    if ( undirected )
    { if (v == w)
        error_handler(1,"new_edge(v,e1,w,e2): selfloop in undirected
graph.");
      if (e1 && v != source(e1) && v != target(e1))

```



```

        error_handler(1,"new_edge(v,e1,w,e2): v is not adjacent to
e1.");
        if (e2 && w != source(e2) && w != target(e2))
            error_handler(1,"new_edge(v,e1,w,e2): w is not adjacent to
e2.");

        v->insert_adj_edge(e,e1,0,0,d1);
        w->insert_adj_edge(e,e2,0,1,d2);
    }
    else
    { if (e1 && v != source(e1))
        error_handler(1,"new_edge(v,e1,w,e2): v is not source of e1.");
      if (e2 && w != source(e2) && w != target(e2))
        error_handler(1,"new_edge(v,e1,w,e2): w is not target of e2.");

        v->insert_adj_edge(e,e1,0,0,d1);
        w->insert_adj_edge(e,e2,1,1,d2);
    }
}

```

```

edge graph::new_edge(node v, edge e1, node w, edge e2, GenPtr i,int
d1,int d2)
{
    // add edge (v,w,i)
    // after(if d1=0)/before(if d1=1) e1 to adj_list of v
    // after(if d2=0)/before(if d2=1) e2 to in_list (adj_list) of w
    // (most general form of new_edge)

    if ( undirected )
    { if (v == w)
        error_handler(1,"new_edge(v,e1,w,e2): selfloop in undirected
graph.");
      if (e1 && v != source(e1) && v != target(e1))
        error_handler(1,"new_edge(v,e1,w,e2): v is not adjacent to
e1.");
      if (e2 && w != source(e2) && w != target(e2))
        error_handler(1,"new_edge(v,e1,w,e2): w is not adjacent to
e2.");
    }
    else
    { if (e1 && v != source(e1))
        error_handler(1,"new_edge(v,e1,w,e2): v is not source of e1.");
      if (e2 && w != source(e2) && w != target(e2))
        error_handler(1,"new_edge(v,e1,w,e2): w is not target of e2.");
    }

    pre_new_edge_handler(v,w);
    edge e = add_edge(v,w,i);
    ins_adj_edge(e,v,e1,w,e2,d1,d2);
    post_new_edge_handler(e);
    return e;
}

```

```

edge graph::new_edge(node v, edge e1, node w, GenPtr i,int d)
{
    // add edge (v,w) after/before e1 to adj_list of v
    // append it to in_list (adj_list) of w

    return new_edge(v,e1,w,nil,i,d,0);
}

```

```

}

edge graph::new_edge(node v, node w, edge e2, GenPtr i,int d)
{
    // append edge (v,w) to adj_list of v
    // insert it after/before e2 to in_list (adj_list) of w

    return new_edge(v,nil,w,e2,i,d,0);
}

edge graph::new_edge(edge e1, node w, GenPtr i, int dir)
{
    // add edge (source(e1),w) after/before e1 to adj_list of source(e1)
    // append it to in_list (adj_list) of w

    return new_edge(source(e1),e1,w,nil,i,dir,0);
}

edge graph::new_edge(node v, edge e2, GenPtr i, int dir)
{
    // append edge(v,target(e2)) to adj_list of v
    // insert it after/before e2 to in_list (adj_list) of target(e2)

    return new_edge(v,nil,target(e2),e2,i,0,dir);
}

edge graph::new_edge(edge e1, edge e2, GenPtr i, int dir1, int dir2)
{
    //add edge (source(e1),target(e2))
    //after(dir=0)/before(dir=1) e1 to adj_list of source(e1)
    //after(dir=1)/before(dir=1) e2 to in_list (adj_list) of target(e2)

    return new_edge(source(e1),e1,target(e2),e2,i,dir1,dir2);
}

edge graph::new_edge(node v, node w, GenPtr i)
{
    // append (v,w) it to adj_list of v and to in_list (adj_list) of w

    return new_edge(v,nil,w,nil,i,0,0);
}

node graph::split_edge(edge e, GenPtr node_inf, edge& e1, edge& e2)
{
    // splits e into e1 and e2 by putting new node v on e

    //node v = source(e);
    node w = target(e);
    node u = add_node(node_inf);

    e1 = e;
    e2 = add_edge(u,w,e->data[0]);

    copy_edge_entry(e2->data[0]);
}

```

```

    if (undirected)
    { u->append_adj_edge(e2,0,0);
      w->insert_adj_edge(e2,e,0,1,0);
      w->del_adj_edge(e,0,1);
      e->term[1] = u;
      u->append_adj_edge(e,0,1);
    }
    else
    { u->append_adj_edge(e2,0,0);
      w->insert_adj_edge(e2,e,1,1,0);
      w->del_adj_edge(e,1,1);
      e->term[1] = u;
      u->append_adj_edge(e,1,1);
    }

    return u;
}

void graph::del_edge(edge e)
{ node v = source(e);
  node w = target(e);

  if (v->owner != this) error_handler(10,"del_edge(e): e is not in G");

  pre_del_edge_handler(e);

  if (is_hidden(e)) restore_edge(e);

  if (e->rev) e->rev->rev = nil;

  del_adj_edge(e,v,w);

  if (parent == 0) clear_edge_entry(e->data[0]);

  e_list.remove(e);
  e_free.append(e);

  graph_map* m;
  forall(m,map_list[1])
  { int i = m->g_index;
    if (i > 0) m->clear_entry(e->data[i]);
  }

  post_del_edge_handler(v,w);
}

void graph::hide_edge(edge e)
{
  if (is_hidden(e))
    error_handler(1,"graph::hide_edge: edge is already hidden.");

  pre_hide_edge_handler(e);

  node v = source(e);
  node w = target(e);

```

```

    del_adj_edge(e,v,w);

    e_list.remove(e);
    h_list.append(e);

    e->id |= 0x80000000;

    post_hide_edge_handler(e);
}

void graph::restore_edge(edge e)
{
    if (!is_hidden(e))
        error_handler(1,"graph::restore_edge: edge is not hidden.");

    pre_restore_edge_handler(e);

    node v = source(e);
    node w = target(e);

    h_list.remove(e);
    e_list.append(e);

    if (undirected)
    {
        v->append_adj_edge(e,0,0);
        w->append_adj_edge(e,0,1);
    }
    else
    {
        v->append_adj_edge(e,0,0);
        w->append_adj_edge(e,1,1);
    }

    e->id = index(e);

    post_restore_edge_handler(e);
}

void graph::restore_all_edges()
{
    edge e = (edge)h_list.head();
    while (e)
    {
        edge succ = (edge)h_list.succ(e);
        restore_edge(e);
        e = succ;
    }
}

void graph::move_edge(edge e,edge e1,edge e2,int d1,int d2)
{
    if (is_hidden(e))
        error_handler(1,"graph::move_edge: cannot move hidden edge.");
    node v0 = source(e);
    node w0 = target(e);
    node v = source(e1);
    node w = target(e2);
    pre_move_edge_handler(e,v,w);
    del_adj_edge(e,source(e),target(e));
    e->term[0] = v;

```

```

    e->term[1] = w;
    ins_adj_edge(e,v,e1,w,e2,d1,d2);
    post_move_edge_handler(e,v0,w0);
}

void graph::move_edge(edge e,edge e1,node w,int dir)
{ if (is_hidden(e))
    error_handler(1,"graph::move_edge:  cannot move hidden edge.");
    node v0 = source(e);
    node w0 = target(e);
    node v = source(e1);
    pre_move_edge_handler(e,v,w);
    del_adj_edge(e,source(e),target(e));
    e->term[0] = v;
    e->term[1] = w;
    ins_adj_edge(e,source(e1),e1,w,nil,dir,0);
    post_move_edge_handler(e,v0,w0);
}

void graph::move_edge(edge e, node v, node w)
{ if (is_hidden(e))
    error_handler(1,"graph::move_edge:  cannot move hidden edge.");
    node v0 = source(e);
    node w0 = target(e);
    pre_move_edge_handler(e,v,w);
    del_adj_edge(e,source(e),target(e));
    e->term[0] = v;
    e->term[1] = w;
    ins_adj_edge(e,v,nil,w,nil,0,0);
    post_move_edge_handler(e,v0,w0);
}

edge graph::rev_edge(edge e)
{ if (is_hidden(e))
    error_handler(1,"graph::move_edge:  cannot move hidden edge.");
    node v = source(e);
    node w = target(e);

    pre_move_edge_handler(e,w,v);

    if (is_hidden(e)) // e hidden
    { e->term[0] = w;
      e->term[1] = v;
      return e;
    }

    if (undirected)
    { edge s = e->succ_adj_edge[0];
      edge p = e->pred_adj_edge[0];
      e->succ_adj_edge[0] = e->succ_adj_edge[1];
      e->pred_adj_edge[0] = e->pred_adj_edge[1];
      e->succ_adj_edge[1] = s;
      e->pred_adj_edge[1] = p;
      e->term[0] = w;
      e->term[1] = v;
    }
    else
    { del_adj_edge(e,v,w);
      e->term[0] = w;
      e->term[1] = v;
    }
}

```

```

        ins_adj_edge(e,w,nil,v,nil,0,0);
    }

    post_move_edge_handler(e,v,w);

    return e;
}

void graph::rev_all_edges()
{ if (!undirected)
  { list<edge> L = all_edges();
    edge e;
    forall(e,L) rev_edge(e);
  }
}

void graph::del_nodes(const list<node>& L)
{ node v;
  forall(v,L) del_node(v);
}

void graph::del_edges(const list<edge>& L)
{ edge e;
  forall(e,L) del_edge(e);
}

void graph::make_undirected()
{
    if (undirected) return;

    list<edge> loops;
    edge e;

    forall_edges(e,*this)
        if (source(e) == target(e)) loops.append(e);

    if ( ! loops.empty() )
        error_handler(0,"selfloops deleted in ugraph constructor");

    forall(e,loops) del_edge(e);

    /* adj_list(v) = out_list(v) + in_list(v) forall nodes v */
    node v;
    forall_nodes(v,*this)
    {
        // append in_list to adj_list

        if (v->first_adj_edge[1] == nil) continue;

        if (v->first_adj_edge[0] == nil) // move in_list to adj_list
        { v->first_adj_edge[0] = v->first_adj_edge[1];
          v->last_adj_edge[0]  = v->last_adj_edge[1];
          v->adj_length[0]     = v->adj_length[1];
        }
        else // both lists are non-empty
        { v->last_adj_edge[0]->succ_adj_edge[0] = v->first_adj_edge[1];

```

```

        v->first_adj_edge[1]->pred_adj_edge[1] = v->last_adj_edge[0];
        v->last_adj_edge[0] = v->last_adj_edge[1];
        v->adj_length[0] += v->adj_length[1];
    }

    v->first_adj_edge[1] = nil;
    v->last_adj_edge[1] = nil;
    v->adj_length[1] = 0;
}

undirected = true;
}

void graph::make_directed()
{
    if (!undirected) return;

    // for every node v delete entering edges from adj_list(v)
    // and put them back into in_list(v)

    node v;
    forall_nodes(v,*this)
    { edge e = v->first_adj_edge[0];
      while (e)
        if (v == target(e))
        { edge e1 = e->succ_adj_edge[1];
          v->del_adj_edge(e,0,1);
          v->append_adj_edge(e,1,1);
          e = e1;
        }
        else
          e = e->succ_adj_edge[0];
    }

    undirected = false;
}

void init_node_data(const graph& G,int i, GenPtr x)
{ node v;
  forall_nodes(v,G) v->data[i] = x;
}

int graph::register_map(graph_map* M)
{ int k = M->kind;

  M->g_loc = map_list[k].append(M);

#ifdef LEDA_GRAPH_DATA
  if (free_data[k].empty())
    error_handler(1,
      string("graph::register_map: all data (%d) slots
used",data_sz[k]));
#endif
}

```

```

    return (free_data[k].empty()) ? -1 : free_data[k].pop();
}

```

```

void graph::unregister_map(graph_map* M)
{ int k = M->kind;
  map_list[k].del_item(M->g_loc);
  if (M->g_index > 0) free_data[k].push(M->g_index);
}

```

```

node graph::choose_node() const
{ int n = number_of_nodes();
  if (n == 0) return nil;
  int r = rand_int(0,n-1);
  node v = first_node();
  while (r-->0) v = succ_node(v);
  return v;
}

```

```

edge graph::choose_edge() const
{ int m = number_of_edges();
  if (m == 0) return nil;
  int r = rand_int(0,m-1);
  edge e = first_edge();
  while (r-->0) e = succ_edge(e);
  return e;
}

```

```

face graph::choose_face() const
{ int l = number_of_faces();
  if (l == 0) return nil;
  int r = rand_int(0,l-1);
  face f = first_face();
  while (r-->0) f = succ_face(f);
  return f;
}

```

```

//-----
// old iterator stuff
//-----

```

```

void graph::init_adj_iterator(node v) const
{ adj_iterator->map_access(v) = nil; }

```

```

bool graph::current_adj_edge(edge& e, node v) const
{ return (e = (edge)adj_iterator->map_access(v)) != nil;}

```

```

bool graph::next_adj_edge(edge& e, node v) const
{ edge cur = (edge)adj_iterator->map_access(v);
  e = (cur) ? adj_succ(cur) : first_adj_edge(v);
  adj_iterator->map_access(v) = e;
  return (e) ? true : false;
}

```



```

    }

    bool graph::next_adj_node(node& w, node v)  const
    { edge e;
      if (next_adj_edge(e,v))
      { w = opposite(v,e);
        return true;
      }
      else return false;
    }

    bool graph::current_adj_node(node& w, node v)  const
    { edge e;
      if (current_adj_edge(e,v))
      { w = opposite(v,e);
        w = target(e);
        return true;
      }
      else return false;
    }

    void graph::reset()  const  // reset all iterators
    { adj_iterator->init(this,max_n_index+1,0);
      node v;
      forall_nodes(v,*this) adj_iterator->map_access(v) = nil;
    }

```

```

/*****
*****
+
+ LEDA 3.5.1
+
+ _planar_map.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****
*****/
#include <LEDA/planar_map.h>

void planar_map::init_face_entries() const
{ face f;
  forall_faces(f,*this) init_face_entry(f->data[0]);
}

void planar_map::copy_face_entries() const
{ face f;
  forall_faces(f,*this) copy_face_entry(f->data[0]);
}

void planar_map::clear_face_entries() const
{ face f;
  forall_faces(f,*this) clear_face_entry(f->data[0]);
}

node planar_map::new_node(const list<edge>& el)
{
  if (el.length() < 2)
    error_handler(1,"planar_map::new_node(el,i):  el.length() < 2.");

  list_item it = el.first();

  edge e0 = el[it];

  it = el.succ(it);

  face f = adj_face(e0);

  edge e;
  forall(e,el)
  { if (adj_face(e) != f)
    error_handler(1,"planar_map::new_node: edges bound different
faces.");
  }

  e = el[it];
}

```

```

it = el.succ(it);

GenPtr face_inf = f->data[0];
copy_face_entry(face_inf);

edge x = new_edge(e0,e);
face fx = adj_face(reversal(x));
clear_face_entry(fx->data[0]);
fx->data[0] = face_inf;

edge e1 = split_edge(x);

while(it)
{ copy_face_entry(face_inf);
  e1 = new_edge(e1,e1[it]);
  face fx = adj_face(reversal(e1));
  clear_face_entry(fx->data[0]);
  fx->data[0] = face_inf;
  it = e1.succ(it);
}

return source(e1);
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _pq_tree.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
+ *****/
//-----
--
// PQ_TREES
//
// R. Hesse, E. Kalliwoda, D. Ambras (1996/97)
//
//-----
--

#include <LEDA/pq_tree.h>
#include <LEDA/queue.h>

const int
  D_NODE_INVALID=0,
  D_NODE_PNODE  =1,
  D_NODE_QNODE  =2,
  D_NODE_LEAF   =3,
  D_NODE_DIR    =4,

  D_NODE_UNMARKED =6,
  D_NODE_UNBLOCKED =7,
  D_NODE_BLOCKED  =8,
  D_NODE_QUEUED   =9,

  D_NODE_EMPTY    =10,
  D_NODE_FULL     =11,
  D_NODE_PARTIAL  =12,
  D_NODE_DOUBLE_PARTIAL =13;

pq_node_struct::pq_node_struct()
{
  leaf_index  = 0;
  child_count = 0;

  parent      = NULL;
  link_one_side = NULL;
  link_other_side = NULL;
  right_most  = NULL;
  left_most   = NULL;

  type        = D_NODE_PNODE;

```

```

    parent_type    = D_NODE_QNODE;

    node_reset();
}

void pq_node_struct::node_reset()
{
    mark    = D_NODE_UNMARKED;
    status = D_NODE_EMPTY;      // vielleicht auf FULL umstellen ***

    pert_leaf_count  = 0;
    pert_child_count = 0;
    full_child_count = 0;

    part_child1 = NULL;
    part_child2 = NULL;
}

pq_tree::pq_tree(int lsize)
{
    successful          = true;
    too_many_part       = false;
    leaves_size         = lsize;
    blocked_chain_count = 0;
    blocked_nodes_count = 0;

    root                = NULL;
    pseudo_root          = new pq_node_struct;
    pseudo_root->leaf_index = 2;

    if (lsize > 0) leaves_init(lsize+1);
    else leaves = NULL;
}

//-----
//
// TEMPLATES
//-----
//

void pq_tree::fix_part_child_direction(pq_node x, bool part_child2_too)
{
    pq_node    k;
    pq_node    child1= x->part_child1;
    pq_node    child2= x->part_child2;

    if (child1->left_most->status == D_NODE_FULL)
    { k= child1->left_most;
      child1->left_most  = child1->right_most;
      child1->right_most = k;
    }

    if (part_child2_too)
        if (child2->right_most->status == D_NODE_FULL)

```

```

    { k= child2->left_most;
      child2->left_most = child2->right_most;
      child2->right_most = k;
    }
}

```

```

inline void pq_tree::append_as_right_child(pq_node parent, pq_node
child)
{
    pq_node l= parent->right_most;

    if (l->link_one_side)
        l->link_other_side = child;
    else
        l->link_one_side = child;

    child->link_one_side = l;
    child->link_other_side= NULL;

    parent->right_most = child;
}

```

```

pq_node pq_tree::hang_down_full_children(pq_node x, pq_node z)
{
    int      x_full_child_count= x->full_child_count;
    pq_node  y;

    if (x_full_child_count == 0) return NULL;

    if (x_full_child_count == 1)
    {
        y= remove_from_siblings(x, x->right_most);
        y->parent = z;
        y->parent_type = z->type;
        append_as_right_child(z, y);

        z->child_count++;
        z->full_child_count++;
        z->pert_leaf_count = x->pert_leaf_count;
        x->full_child_count = 0;
        x->child_count--;

        return y;
    }
}

```

```

pq_node l;

y = new pq_node_struct;
y->parent = z;
y->parent_type = z->type;
y->status = D_NODE_FULL;
processed.push(y);

```

```

y->left_most = x->right_most;
y->right_most = x->right_most;

y->left_most->parent = y;
l = NULL;
for(int i = 1; i < x_full_child_count; i++)
{ go_to_sibling(y->left_most, l);
  y->left_most->parent = y;
}

pq_node k, sibling;
pq_node y_left = y->left_most;

if (y_left->link_one_side == l)
{ k = y_left->link_other_side;
  y_left->link_other_side = NULL;
}
else
{ k = y_left->link_one_side;
  y_left->link_one_side = NULL;
}

if (x != z)
{ x->right_most = k;
  append_as_right_child(z, y);
  sibling = NULL;
}
else
{ x->right_most = y;
  y->link_one_side = k;
  sibling = y;
}

if (k->link_one_side == y_left)
  k->link_one_side = sibling;
else
  k->link_other_side = sibling;

y->child_count = y->full_child_count = x->full_child_count;

x->full_child_count = 0;
x->child_count -= y->child_count;

y->pert_leaf_count = x->pert_leaf_count;
if (x->part_child1)
  y->pert_leaf_count -= x->part_child1->pert_leaf_count;
if (x->part_child2)
  y->pert_leaf_count -= x->part_child2->pert_leaf_count;

z->child_count++;
z->full_child_count++;
z->pert_leaf_count += y->pert_leaf_count;

return y;
}

```

```

pq_node pq_tree::remove_from_siblings(pq_node parent, pq_node child)
{
    if (child->link_one_side)
        if (child->link_one_side->link_one_side == child)
            child->link_one_side->link_one_side = child->link_other_side;
        else
            child->link_one_side->link_other_side = child->link_other_side;

    if (child->link_other_side)
        if (child->link_other_side->link_one_side == child)
            child->link_other_side->link_one_side = child->link_one_side;
        else
            child->link_other_side->link_other_side = child->link_one_side;

    if (parent->right_most == child)
        parent->right_most = child->link_one_side ?
            child->link_one_side : child->link_other_side;

    if (parent->left_most == child)
        parent->left_most = child->link_one_side ?
            child->link_one_side : child->link_other_side;

    if (parent->part_child1 == child)
    { parent->part_child1 = parent->part_child2;
      parent->part_child2 = NULL;
    }
    if (parent->part_child2 == child) parent->part_child2 = NULL;

    return child;
}

```

```

void pq_tree::replace_in_siblings(pq_node x, pq_node z)
{
    z->parent = x->parent;
    z->parent_type = x->parent_type;

    z->link_one_side = x->link_one_side;
    z->link_other_side = x->link_other_side;

    if (z->link_one_side)
        if (z->link_one_side->link_one_side == x)
            z->link_one_side->link_one_side = z;
        else
            z->link_one_side->link_other_side = z;

    if (z->link_other_side)
        if (z->link_other_side->link_one_side == x)
            z->link_other_side->link_one_side = z;
        else
            z->link_other_side->link_other_side = z;

    if (z->parent)
    { if (z->parent->right_most == x) z->parent->right_most = z;
    }
}

```



```

    if (z->parent->left_most == x) z->parent->left_most = z;
}

bool pq_tree::set_as_part_child(pq_node x)
{
    x->status = D_NODE_PARTIAL;

    if (x->parent->part_child1)
    { if (x->parent->part_child2)
      { too_many_part = true;
        return false;
      }
      else x->parent->part_child2 = x;
    }
    else x->parent->part_child1 = x;

    return true;
}

inline void pq_tree::append_as_left_child(pq_node parent, pq_node child)
{
    pq_node l= parent->left_most;

    child->parent = parent;
    child->parent_type = D_NODE_QNODE; // ist immer so

    if (l->link_one_side)
        l->link_other_side = child;
    else
        l->link_one_side = child;

    child->link_one_side = NULL;
    child->link_other_side= l;

    parent->left_most = child;
}

void pq_tree::delete_part_node_parent(pq_node x)
{
    replace_in_siblings(x, x->part_child1);

    if (x == root)
    { root= x->part_child1;
      root->parent = NULL;
      root->link_one_side = NULL;
      root->link_other_side = NULL;
    }

    delete x;
}

```

```

bool pq_tree::template_PQL1(pq_node x, bool is_pseudo_root)
{
#ifdef _DEBUG_PQ_TREE
    if (x->type == D_NODE_PNODE) cout << "P1";
    else
        if (x->type == D_NODE_QNODE) cout << "Q1";
        else cout << "L1";
#endif

    if (x->type == D_NODE_QNODE)
    { pq_node k;
      pq_node last_k= NULL;

      if (x->part_child1) return false;
      if (x == pseudo_root) return true;

      k = x->right_most;
      while (k)
      { if (k->type != D_NODE_DIR && k->status == D_NODE_EMPTY)
          return false;
        go_to_sibling(k, last_k);
      }
    }
    else
        if (x->full_child_count != x->child_count) return false;

    if (x->parent_type == D_NODE_PNODE && x->parent &&
        x->parent->right_most != x )
    {
        remove_from_siblings(x->parent, x);
        append_as_right_child(x->parent, x);
    }

    x->status = D_NODE_FULL;
    processed.push(x);

    if (is_pseudo_root)
    {
        pseudo_root->left_most = x;
        pseudo_root->right_most = x;
    }
    else
        if (x->parent) x->parent->full_child_count++;

    return true;
}

bool pq_tree::template_P3(pq_node x, bool is_pseudo_root)
{
    pq_node y, z;

```

```

#ifdef _DEBUG_PQ_TREE
    if (is_pseudo_root) cout << "P2";
    else cout << "P3";
    assert(x != root);
    cons_pq_tree(root, "b", "P2/3");
#endif

    if (x->part_child1) return false;

    if (x->full_child_count > 1)
        y= hang_down_full_children(x, x);
    else
        y= x->right_most;

    if (is_pseudo_root)
    {
        pseudo_root->left_most = y;
        pseudo_root->right_most = y;
        x->node_reset();

        return true;
    }

    if (x->child_count > 2)
    {
        z = new pq_node_struct;
        z->mark = D_NODE_UNBLOCKED;

        //for correct blocked chain handling in reduce():

        if (x == pseudo_root->left_most) pseudo_root->left_most = z;
        if (x == pseudo_root->right_most) pseudo_root->right_most = z;

        replace_in_siblings(x, z);

        x->right_most = y->link_one_side ?
            y->link_one_side : y->link_other_side;

        if (x->right_most->link_one_side == y)
            x->right_most->link_one_side = NULL;
        else
            x->right_most->link_other_side = NULL;

        x->child_count--;
        x->node_reset();

        x->parent = y->parent = z;
        z->left_most = x;
        z->right_most = y;

        x->link_one_side = NULL;
        x->link_other_side= y;

        y->link_one_side = NULL;
        y->link_other_side= x;
    }

```

```

    z->pert_leaf_count = x->pert_leaf_count;
    z->full_child_count = 1;
    z->child_count = 2;
}
else z = x;

z->type = D_NODE_QNODE;
z->left_most->parent_type = D_NODE_QNODE;
z->right_most->parent_type = D_NODE_QNODE;

return set_as_part_child(z);
}

```

```

bool pq_tree::template_P5(pq_node x, bool is_pseudo_root)
{
    pq_node y, k, z;

#ifdef _DEBUG_PQ_TREE
    if (is_pseudo_root) cout << "P4";
    else cout << "P5";
    assert(x != root);
    cons_pq_tree(root, "b", "P4/5");
#endif

    if (x->part_child2 || too_many_part ) return false;

    // if (!x->part_child1 || x->part_child2 || too_many_part )
    // return false;
    // "|| too_many_part" muss bleiben, x->part_child1 darf nicht !?

    fix_part_child_direction(x);

    z = x->part_child1;

    y= hang_down_full_children(x, z);      // Teste, ob es volle Kinder
    gibt ? ***                             // ( sonst y= NULL )
                                           // it's _P4
    if (is_pseudo_root)
    {
        pseudo_root->right_most = y;

        // suche letztes volles Kind:

        k = y->link_one_side ?
            y->link_one_side : y->link_other_side;

        while (k->status == D_NODE_FULL || k->type == D_NODE_DIR)
            go_to_sibling(k, y);

        pseudo_root->left_most = y;        // y= letztes volles Kind

        z->node_reset();

        if (x->child_count == 1)

```

```

    delete_part_node_parent(x);
else
    x->node_reset();

return true;
}

if (x->child_count > 2)
{
    z->pert_leaf_count = x->pert_leaf_count;
    x->child_count--;
    x->pert_leaf_count = 0;

    x->part_child1 = NULL;
    remove_from_siblings(x, z);
    replace_in_siblings(x, z);
    set_as_part_child(z);
    append_as_left_child(z, x);

    x->node_reset();
}

else
{
    if (x->child_count == 2)
    {
        y = (x->right_most == z) ?
            x->left_most : x->right_most;
        append_as_left_child(z, y);
    }

    // for correct blocked chain handling in reduce():

    z->mark = D_NODE_UNBLOCKED;
    if (x == pseudo_root->left_most) pseudo_root->left_most = z;
    if (x == pseudo_root->right_most) pseudo_root->right_most = z;

    replace_in_siblings(x, z);
    set_as_part_child(z);

    delete x;
}

return !too_many_part;
}

bool pq_tree::template_P6(pq_node x)
{
    #ifdef _DEBUG_PQ_TREE
        cout << "P6";
    #endif
}

```

```

pq_node k, l;

if (!x->part_child2 || too_many_part) return false;

fix_part_child_direction(x, true);
hang_down_full_children(x, x->part_child1);

l = x->part_child1->right_most;
k = l->link_one_side ?
    l->link_one_side : l->link_other_side;

while (k->status == D_NODE_FULL || k->type == D_NODE_DIR)
    go_to_sibling(k, l);
pseudo_root->left_most = l;          // linkestes volles Kind von x-
>part1
l->parent = x->part_child1;

l = x->part_child2->left_most;
k = l->link_one_side ?
    l->link_one_side : l->link_other_side;

while (k->status == D_NODE_FULL || k->type == D_NODE_DIR)
    go_to_sibling(k, l);
pseudo_root->right_most = l;        // rechtestes volles Kind von x-
>part2

k = x->part_child1;
l = remove_from_siblings(x, x->part_child2);
x->child_count--;

if (k->right_most->link_one_side)
    k->right_most->link_other_side = l->left_most;
else
    k->right_most->link_one_side = l->left_most;

if (l->left_most->link_one_side)
    l->left_most->link_other_side = k->right_most;
else
    l->left_most->link_one_side = k->right_most;

k->right_most = l->right_most;
k->right_most->parent = k;

delete l;

x->part_child1->node_reset();

if (x->child_count == 1)
    delete_part_node_parent(x);
else
    x->node_reset();

return true;
}

```

```

bool pq_tree::template_Q2(pq_node x)
{
    // template for a Q-node with empties and/or 1 partial
    child

#ifdef _DEBUG_PQ_TREE
    cout << "Q2" << endl;
    if (pseudo_root->leaf_index > 1) show("in Q2",x);
    //show("in Q2",x);
    //show("in Q2",root);
#endif

    if (x->part_child2) return false;

    // if (!x->part_child1 || x->part_child2 || too_many_part) // darf
    nicht !
    // return false;

    pq_node k; // to run through the children
    pq_node l; // dito
    pq_node l1,l2; // dito
    pq_node m,n; // dito, but see text
    pq_node p1=NULL; // the part_child
    pq_node d1; // the side the full's will be turned
to ...
    pq_node d2; // and the other side
    pq_node aux_leftm=NULL;
    pq_node aux_rightm=NULL; // dummies for pseudo_root->end_most's

    bool full_found=false; // turn the full's to the outside
    bool a,b,c,d; // for testing of blocked chain

    if (p1 = x->part_child1)
    {
        l1 = l2 = x->part_child1;

        m = p1->link_one_side;
        skip_dir(m,l1);

        n = p1->link_other_side;
        skip_dir(n,l2);

        if (x->full_child_count)
        {
            a = m ? m->status == D_NODE_FULL : 0;
            b = n ? n->status == D_NODE_FULL : 0;

            if (!(a ^ b)) return false; // xor; (both (full)) or (both (NULL
or empty))

            if (x->full_child_count == 1)
            {

```

```

    if (a)
    { d1 = p1->link_one_side;
      d2 = p1->link_other_side;
      k = m;
      l = l1;
    }
    else
    { d2 = p1->link_one_side;
      d1 = p1->link_other_side;
      k = n;
      l = l2;
    }

    go_to_sibling(k, l);

    skip_dir(k, l);
    aux_leftm = l;

// in case of a blocked chain the end_most's of pseudo_root are already
used
// and valid
// next reduce round Q2 will match pseudo_root as the father of the
blocked chain

    full_found = true;

}
else
{
    if (a)
    { k = m; l = l1; }
    else
    { k = n; l = l2; }

    for(int i = 2; i <= x->full_child_count; i++)
    { go_to_sibling(k, l);
      skip_dir(k,l);
      if (!k || k->status != D_NODE_FULL ) return false;

      //there is an empty between the full's
      //or not all full's are at one side
    }

    full_found = true;
    go_to_sibling(k, l);

    skip_dir(k,l);
    aux_leftm = l;

    if (a)
    { d1 = p1->link_one_side; d2 = p1->link_other_side; }
    else
    { d2 = p1->link_one_side; d1 = p1->link_other_side; }

}

}
else
{
    //if (x->full_child_count)

    //... no full's, only a part_child

    if (pseudo_root->status == D_NODE_FULL)

```



```

{ // ROOT(T,S) is reached

    d1 = p1->link_one_side;
    d2 = p1->link_other_side;

    if (d1->type == D_NODE_DIR)
    { l = p1;
      k = d1;
      skip_dir(k,l);
      aux_leftm = l;
    }
    else
        if (p1->right_most->status == D_NODE_FULL)
            aux_leftm = p1->right_most;
        else
            aux_leftm = p1->left_most;
    }
    else
    { if (m && n) return false;          //-part_child is between
empties

        d1 = NULL;                      //-part_child's full endmost
will be
        d2 = m ? p1->link_one_side : p1->link_other_side;
                                //-turned outside and becomes an
end_most
    }

} //end of "if (p1 = x->part_child1)"
else
{ // there's no part_child

    if (x->left_most->status == D_NODE_FULL)
        l = x->left_most;
    else
    { l = x->right_most;
      if ( l->status != D_NODE_FULL) return false;
    }

//    if ( pseudo_root->type != D_NODE_QNODE || x == pseudo_root)
//        aux_rightm = aux_leftm = l;
//        aux_rightm = aux_leftm = l;

    if (x->full_child_count > 1)
    {
        k = l->link_one_side ? l->link_one_side : l->link_other_side;
        skip_dir(k,l);

        if (k->status == D_NODE_FULL)
        { for(int i = 2; i <= x->full_child_count; i++)
            { skip_dir(k,l);
              if ( k->status != D_NODE_FULL) return false;
              go_to_sibling(k, l);
            }

//            if ( pseudo_root->type != D_NODE_QNODE || x == pseudo_root )
//            if ( pseudo_root->status == D_NODE_FULL)
//            {
//                skip_dir(k,l);
//                aux_leftm = l;
//            }

```

```

    }
    else return false;    // empties are intermingled with full's
} //else from "if (p1 = x->part_child1)"
                                //no empty's and no part_child between full's
                                //template applicable, now the replacement:
if (p1)
{
    if (p1->right_most->status == D_NODE_FULL)
    { m = p1->right_most; n = p1->left_most; }
    else
    { m = p1->left_most, n = p1->right_most; }

    if (m->link_one_side)
        m->link_other_side = d1;
    else
        m->link_one_side = d1;

    if (d1)
    {
        if (d1->link_one_side == p1)
            d1->link_one_side = m;
        else
            d1->link_other_side = m;
    }

    if (n->link_one_side)
        n->link_other_side = d2;
    else
        n->link_one_side = d2;

    if (d2)
    {
        if (d2->link_one_side == p1)
            d2->link_one_side = n;
        else
            d2->link_other_side = n;
    }

    if (p1 == x->left_most)
    { x->left_most = full_found ? n : m;
      x->left_most->parent = x;
    }

    if (p1 == x->right_most)
    { x->right_most = full_found ? n : m;
      x->right_most->parent = x;
    }

    x->part_child1 = NULL;

    //reversed (if necessary) and chained

    x->full_child_count += p1->full_child_count;
    delete p1;

    l = d1;
    k = m;

    while (k && k->status == D_NODE_FULL)
    { k->parent = x;
      go_to_sibling(k, l);
      skip_dir(k,l);
    }
}

```

```

    aux_rightm = 1;

}

if (aux_leftm && aux_rightm)
{ c = aux_leftm->link_one_side && aux_leftm->link_other_side;
  d = aux_rightm->link_one_side && aux_rightm->link_other_side;
  if (c && d) x->status = D_NODE_DOUBLE_PARTIAL;
}

if (pseudo_root->status == D_NODE_FULL || x == pseudo_root){
    pseudo_root->left_most = aux_leftm;
    pseudo_root->right_most = aux_rightm;
}
else
if (x->status == D_NODE_DOUBLE_PARTIAL) return false;

if (x != pseudo_root)
{ // father pointer of x is valid, that means != NULL
  if (pseudo_root->status == D_NODE_FULL) // ROOT(T,S) reached
    x->node_reset();
  else
  { x->status = D_NODE_PARTIAL;
    if (x->parent->part_child1)
      { if (x->parent->part_child2)
          { too_many_part = true;
            return false;
          }
        else
          x->parent->part_child2 = x;
      }
    else
      x->parent->part_child1 = x;
    //x becomes one of the x->parent->part_children
  }
}

return true;
}

```

```

bool pq_tree::template_Q3(pq_node x)
{ // template for a Q-node with exactly 2 partial
  children

```

```

#ifdef _DEBUG_PQ_TREE
cout << "Q3";
#endif

```

```

if ( pseudo_root->status != D_NODE_FULL ||
    too_many_part ||
    !x->part_child2 ) return false;

```

```

pq_node k;      //to run through the children
pq_node l;      //dito
pq_node p1;     //a part_child ...
pq_node d1;     //... and its neighbour in direction to the full's
pq_node p2;     //dito
pq_node d2;     //dito
pq_node mm,m;   //to run through the children
pq_node nn,n;   //dito
pq_node l1,l2;  //dito

unsigned char cc;    //for a check

bool a,b;

l1 = l2 = p1 = x->part_child1;
m = p1->link_one_side;
skip_dir(m,l1);

n = p1->link_other_side;
skip_dir(n,l2);

l1 = l2 = p2 = x->part_child2;
mm = p2->link_one_side;
skip_dir(mm,l1);
nn = p2->link_other_side;
skip_dir(nn,l2);

if (x->full_child_count)
{ // x has full children

    cc = m ? (m->status == D_NODE_PARTIAL ? 1 :
              (m->status == D_NODE_FULL ? 2 : 0)) : 0;
    cc += n ? (n->status == D_NODE_PARTIAL ? 1 :
              (n->status == D_NODE_FULL ? 2 : 0)) : 0;
    if (cc != 2) return false; // OH GOTT !! ***

    d1 = (m && m->status == D_NODE_FULL) ? p1->link_one_side : p1->
>link_other_side;

    cc = mm ? (mm->status == D_NODE_PARTIAL ? 1 : (mm->status ==
D_NODE_FULL ? 2 : 0)) : 0;
    cc += nn ? (nn->status == D_NODE_PARTIAL ? 1 : (nn->status ==
D_NODE_FULL ? 2 : 0)) : 0;
    if (cc != 2) return false; // ***

    d2 = (mm && mm->status == D_NODE_FULL) ? p2->link_one_side : p2->
>link_other_side;

    /*
    explanation:
    check values for
    NULL empty part full
    0      0      1      2

```

One sibling is "NULL" or "empty" and the other is "full" is a necessary condition here for a valid Q3 situation (the rest of the test follows).

```

*/
if (x->full_child_count > 1)
{
    if (mm && mm->status == D_NODE_FULL)
    { d2 = p2->link_one_side; k = mm; l = 11;}
    else
    { d2 = p2->link_other_side; k = nn; l = 12;}

    for(int i = 2; i <= x->full_child_count; i++)
    { go_to_sibling(k, l);
      skip_dir(k,l);

      if (!k || k->status != D_NODE_FULL) return false;
      // because there is an empty or an part_child between the
full's
    }

    go_to_sibling(k, l);
    skip_dir(k,l);

    if (!k || k->status != D_NODE_PARTIAL) return false;
}
}
else
{ // x has no full child

    a = m ? m->status == D_NODE_PARTIAL : 0;
    b = n ? n->status == D_NODE_PARTIAL : 0;
    d1 = a ? p1->link_one_side : p1->link_other_side;
    if (!(a ^ b)) return false;

    a = mm ? mm->status == D_NODE_PARTIAL : 0;
    b = nn ? nn->status == D_NODE_PARTIAL : 0;
    d2 = a ? p2->link_one_side : p2->link_other_side;
    if (!(a ^ b)) return false;
}

//no empties and no part_child between full's
//template is applicable, now the replacement:

pq_node m1,m2; //the full end_most of a part_child
pq_node n1,n2; //the empty end_most of a part_child
pq_node o;      //a dummy

if (d1 == p2)
{ // the partial children are neighbours

    if (p1->right_most->status == D_NODE_FULL)
    { m1 = p1->right_most;
      n1 = p1->left_most;
    }
    else
    { m1 = p1->left_most;
      n1 = p1->right_most;
    }

    if (p2->right_most->status == D_NODE_FULL)
    { m2 = p2->right_most;
      n2 = p2->left_most;
    }
}

```

```

else
{ m2 = p2->left_most;
  n2 = p2->right_most;
}

o = (p1->link_one_side == d1) ? p1->link_other_side : p1-
>link_one_side;

if (m1->link_one_side) m1->link_other_side = m2; else m1-
>link_one_side = m2;
if (m2->link_one_side) m2->link_other_side = m1; else m2-
>link_one_side = m1;
if (n1->link_one_side) n1->link_other_side = o; else n1-
>link_one_side = o;

if (o)
if (o->link_one_side == p1)
o->link_one_side = n1;
else
o->link_other_side = n1;

o = (p2->link_one_side == d2) ? p2->link_other_side : p2-
>link_one_side;

if (n2->link_one_side)
n2->link_other_side = o;
else
n2->link_one_side = o;

if (o)
if (o->link_one_side == p2)
o->link_one_side = n2;
else
o->link_other_side = n2;

d1 = m2;
d2 = m1;

//reversed and chained
}
else
{ // full children between the partial

if (p1->right_most->status == D_NODE_FULL)
{ m1 = p1->right_most;
  n1 = p1->left_most;
}
else
{ m1 = p1->left_most;
  n1 = p1->right_most;
}

o = (p1->link_one_side == d1) ? p1->link_other_side : p1-
>link_one_side;

if (m1->link_one_side) m1->link_other_side = d1; else m1-
>link_one_side = d1;
if (d1->link_one_side == p1) d1->link_one_side = m1; else d1-
>link_other_side = m1;
if (n1->link_one_side) n1->link_other_side = o; else n1-
>link_one_side = o;

```

```

if (o)
    if (o->link_one_side == p1)
        o->link_one_side = n1;
    else
        o->link_other_side = n1;

if (p2->right_most->status == D_NODE_FULL)
    { m2 = p2->right_most;
      n2 = p2->left_most;
    }
else
    { m2 = p2->left_most;
      n2 = p2->right_most;
    }

o = (p2->link_one_side == d2) ? p2->link_other_side : p2-
>link_one_side;

    if (m2->link_one_side)          m2->link_other_side = d2; else m2-
>link_one_side = d2;
    if (d2->link_one_side == p2) d2->link_one_side = m2; else d2-
>link_other_side = m2;
    if (n2->link_one_side)          n2->link_other_side = o; else n2-
>link_one_side = o;

if (o)
    if (o->link_one_side == p2)
        o->link_one_side = n2;
    else
        o->link_other_side = n2;

// reversed and chained
}

l = d1;
k = m1;

while (k->status == D_NODE_FULL)
{ go_to_sibling(k, l);
  skip_dir(k,l);
}

pseudo_root->left_most = l;

l = d2;
k = m2;

while (k->status == D_NODE_FULL)
{ go_to_sibling(k, l);
  skip_dir(k,l);
}

pseudo_root->right_most = l;          //end_most of pseudo_root now is
valid

if (p1 == x->left_most)
    { x->left_most = n1;
      n1->parent = x;
    }
else

```

```

        if (p2 == x->left_most)
        { x->left_most = n2;
          n2->parent = x;
        }

        if (p1 == x->right_most)
        { x->right_most = n1;
          n1->parent = x;
        }
        else
        { if (p2 == x->right_most)
          { x->right_most = n2;
            n2->parent = x;
          }
        }

        if (x != pseudo_root) x->node_reset();

        pseudo_root->left_most->parent = x;

        delete p1;
        delete p2;

        x->part_child1 = x->part_child2 = NULL;

        return true;
    }

```

```

pq_tree::~pq_tree()
{
    if (!successful) del_subtree(root);
    delete pseudo_root;
    delete[] leaves;
}

```

```

bool pq_tree::reduction(list<int>& S)
{
    list_item lit= processed.first();

    if (lit)
    { do processed.inf(lit)->node_reset();
      while( lit= processed.succ(lit) );

      processed.clear();
    }

    pseudo_root->type = D_NODE_PNODE;
    pseudo_root->node_reset();

    successful = bubble(S) && reduce(S);
}

```



```

#ifdef _DEBUG_PQ_TREE
    if (!successful) show("not successful",root);
#endif

return successful;
}

inline void pq_tree::frontier(list<int> &F)
{ F.clear();
  sequence(F, root);
}

void pq_tree::sequence(list<int>& S, pq_node x, pq_node l)
{
    if (x->type == D_NODE_LEAF)
    { S.append(x->leaf_index);
      return;
    }

    pq_node k = x->left_most;
    pq_node r = x->right_most;

    do
    { if (k->type == D_NODE_DIR)
      { int i = S.pop() + 1; // insert one more DIR-ptr in the
sequence S // with respect to its direction

        if (k->link_one_side == 1)
            S.push(k->leaf_index);
        else
            S.push(-k->leaf_index);
        S.push(i);
        processed.push(k); // we have to delete the DIR-ptr in
update()
    }
    else
        sequence(S, k);

    go_to_sibling(k, l);
  } while (l != r);
}

bool pq_tree::bubble(list<int>& S)
{
    queue<pq_node> Q;
    pq_node x, y, z, k, l;
    int m, blocked_found;

#ifdef _DEBUG_PQ_TREE

```

```

cout << endl;
#endif

root_reached = 0;
blocked_chain_count = 0;
blocked_nodes_count = 0;

pseudo_root->left_most = NULL;
pseudo_root->right_most = NULL;

forall(m, S)
{ x = leaves[m];
  x->mark = D_NODE_QUEUED;  Q.append(x);
}

#ifdef _DEBUG_PQ_TREE
printf(" %d ", this->pseudo_root->leaf_index);
cons_pq_tree(root, "a", "bubble");
// show("in bubble", root);
#endif

while ((Q.size() + blocked_chain_count + root_reached) > 1)
{
  if (Q.empty()) return false;

  x = Q.pop();

  if ( x->parent_type == D_NODE_PNODE || !x->link_one_side || !x-
>link_other_side)
    x->mark = D_NODE_UNBLOCKED;
  else
  { // try to make it valid in constant time
    x->mark = D_NODE_BLOCKED;
    k = x->link_one_side; l = x;
    skip_dir(k, l);
    if ( k->mark == D_NODE_UNBLOCKED)
    { //the link_one_side_sibling has a valid parent
      x->mark = D_NODE_UNBLOCKED;
      x->parent = k->parent;
    }
    else
    { k = x->link_other_side;  l = x;
      skip_dir(k, l);
      if ( k->mark == D_NODE_UNBLOCKED)
      { //the link_other_side_sibling has a valid parent
        x->mark = D_NODE_UNBLOCKED;
        x->parent = k->parent;
      }
    }
  }

  if (x->mark == D_NODE_UNBLOCKED)
  { //x has got a valid parent
    y = x->parent;
    z = x->link_one_side;
    if ( z && (z->mark == D_NODE_BLOCKED || z->type == D_NODE_DIR) )
    { l = x;
      blocked_found = 0;
      while ( z && (z->mark == D_NODE_BLOCKED || z->type == D_NODE_DIR)
)

```

```

        { if ( z->type != D_NODE_DIR)
          { blocked_found = 1;
            z->parent = y;
            z->mark = D_NODE_UNBLOCKED;
            y->pert_child_count++;
            blocked_nodes_count--;
          }

          go_to_sibling(z, 1);
        }
        if (blocked_found) blocked_chain_count--;
    } //to unblock a blocked chain in x->link_one_side
direction

    z = x->link_other_side;

    if ( z && (z->mark == D_NODE_BLOCKED || z->type == D_NODE_DIR) )
    { l = x;
      blocked_found = 0;
      while ( z && (z->mark == D_NODE_BLOCKED || z->type == D_NODE_DIR) )
      { if ( z->type != D_NODE_DIR)
        { blocked_found = 1;
          z->mark = D_NODE_UNBLOCKED;
          z->parent = y;
          y->pert_child_count++;
          blocked_nodes_count--;
        }
        go_to_sibling(z, 1);
      }
      //to unblock a blocked chain in x->link_other_side
direction
      if (blocked_found) blocked_chain_count--;
    }

    if ( !y )
        root_reached = 1;
    else
    { y->pert_child_count++;
      if (y->mark == D_NODE_UNMARKED)
      { Q.append(y);
        y->mark = D_NODE_QUEUED;
      }
    }

    }
else
{ // x's parent is not valid
  k = x->link_one_side;
  l = x;
  skip_dir(k,l);
  if (k->mark == D_NODE_BLOCKED) blocked_chain_count--;
  k = x->link_other_side;
  l = x;
  skip_dir(k,l);
  if (k->mark == D_NODE_BLOCKED) blocked_chain_count--;
  blocked_chain_count++;
  blocked_nodes_count++;
}

} //end of "while (Q->size() + blocked_chain_count + root_reached > 1)"

```

```

    if (blocked_chain_count)
    {
        pseudo_root->pert_child_count = blocked_nodes_count;
        pseudo_root->type = D_NODE_QNODE;

        #ifdef _DEBUG_PQ_TREE
        // show("blocked_chain_count am ende von bubble", root);
        #endif

    }

    #ifdef _DEBUG_PQ_TREE
    //if (pseudo_root->leaf_index > 40) show("Ende bubble", root);
    #endif

    return true;
}

void pq_tree::bubble_reset(pq_node x)
{
    // in case bubble affects the nodes over the pertinent subtree root
    // we have to reset their pert_child_counts

    pq_node k;

    // while (x && x->pert_child_count)          // alt

    while (x && x->pert_child_count && x != pseudo_root)
    {
        if (x->mark == D_NODE_UNBLOCKED) k= x->parent;
        else k= NULL;
        x->pert_child_count = 0;
        x->mark = D_NODE_UNMARKED;
        x->status= D_NODE_EMPTY;
        x = k;
    }
}

bool pq_tree::reduce(list<int>& S)
{
    queue<pq_node> Q;
    pq_node x, y, k, l;

    int S_size = S.size();
    int x_type;

    //show_pq_tree_test(root);

    while (!S.empty())
    { x = leaves[S.pop()];
      x->pert_leaf_count = 1;
      Q.append(x);
    }
}

```

```

    }

    while (!Q.empty())
    { x = Q.pop();

#ifdef _DEBUG_PQ_TREE
    cons_pq_tree(root, "b", "reduce");
    int    aaa=0;
#endif

    if (x->mark == D_NODE_BLOCKED)
    { // a blocked chain exists and its members
      // get the "auxiliary" parent pseudo_root

      x->parent = pseudo_root;
      x->mark = D_NODE_UNBLOCKED;

      l = x;
      k = x->link_one_side;
      skip_dir(k, l);

      if (!k || k->mark == D_NODE_UNMARKED)
      { if (pseudo_root->left_most)
        pseudo_root->right_most = l;
        else
        pseudo_root->left_most = l;
      }

      l = x;
      k = x->link_other_side;
      skip_dir(k, l);

      if (!k || k->mark == D_NODE_UNMARKED)
      { if (pseudo_root->left_most)
        pseudo_root->right_most = l;
        else
        pseudo_root->left_most = l;
      }

      if (Q.empty()) pseudo_root->type = D_NODE_QNODE;
    }

#ifdef _DEBUG_PQ_TREE
    //if (!aaa && x->type != D_NODE_LEAF) show("in reduce, der momentane
    Unterbaum", x);
    if (pseudo_root->leaf_index > 40) show("in reduce, der gesamte Baum",
    root);
    cout << flush;
#endif

    x_type= x->type;                // Type may change in template
    application

    if (x->pert_leaf_count < S_size)
    {
      if (x != pseudo_root)          // ist immer true hier ? !
      { y = x->parent;
        y->pert_leaf_count += x->pert_leaf_count;
        if (!(--y->pert_child_count)) Q.append(y);
      }
    }

```

```

    if (x_type == D_NODE_LEAF)
        if (!template_PQL1(x, false)) return false;

    if (x_type == D_NODE_PNODE)
        if (!template_PQL1(x, false))
            if (!template_P3(x, false))
                if (!template_P5(x, false)) return false;

    if (x_type == D_NODE_QNODE)
        if (!template_PQL1(x, false))
            if (!template_Q2(x)) return false;
}
else
{
    // x is pruned pert subtree root
    (PRUNED(T,S))
    pseudo_root->status = D_NODE_FULL;           // ROOT(T,S) reached
    if (x->parent) bubble_reset(x->parent);

    if (x_type == D_NODE_LEAF)
        if (!template_PQL1(x, true)) return false;

    if (x_type == D_NODE_PNODE)
        if (!template_PQL1(x, true))
            if (!template_P3(x, true))
                if (!template_P5(x, true))
                    if (!template_P6(x)) return false;

    if (x_type == D_NODE_QNODE)
        if (!template_PQL1(x, true))
            if (!template_Q2(x))
                if (!template_Q3(x)) return false;
}

} // while Q not empty

#ifdef _DEBUG_PQ_TREE
cons_pq_tree(root, "b", "reduce 2");
//show("in reduce, der gesamte Baum nach Reduction", root);
#endif

return true;
}

```

```

void pq_tree::pert_sequence(list<int> &S)
{
#ifdef _DEBUG_PQ_TREE
    if (!pseudo_root->left_most){
        cout << "pseudo_root->left_most gleich NULL in pert_sequence" <<
endl;
        exit(1);
    }
#endif

    pq_node l = pseudo_root->left_most;

```

```

pq_node k;

S.clear();
S.push(0);          // preparation

if (l == pseudo_root->right_most)
    // if pseudo_root has only one child life is
easy.
{ sequence(S, l);
  return;
}

    // Otherwise we have to find the direction to pseudo_root-
>right_most.
    // Note that an endmost of pseudo_root not necessary has any
NULL-link.

if (l->link_one_side)
{ k = l->link_one_side;
  skip_dir(k,l);
  l = (k && k->status == D_NODE_FULL)
      ? pseudo_root->left_most->link_other_side
      : pseudo_root->left_most->link_one_side;
}
else
  l = NULL;

    // The direction is detected, we can scan the sequence S.
sequence(S, pseudo_root, l);
}

void pq_tree::leaves_double()
{
    // if any leaf_index > leaves_size occurs we double the size of
array
    // leaves, copy the content of the old array in the lower half
of
    // the new and delete the old. Initially leaves_size is 16.
    // Maybe the user had told the total number of leaves while
defining
    // his PQ tree.

if (leaves_size)
{ int i;

  pq_node* A = new pq_node[2 * leaves_size];
  if (!A) error_handler(1,"pq_tree: out of memory");

  for (i=0; i < leaves_size; i++) A[i] = leaves[i];
  leaves_size *= 2;
  while (i < leaves_size) A[i++] = NULL;
  delete[] leaves;
  leaves = A;
}
else leaves_init(32);

return;
}

```

```
}
```

```
void pq_tree::leaves_init(int lsize)
{
    leaves_size = lsize;

    leaves = new pq_node[leaves_size];
    if (!leaves) error_handler(1, "pq_tree: out of memory");
    for(int i = 0; i < leaves_size; i++) leaves[i] = NULL;
}
```

```
void pq_tree::del_pert_subtree()
{
    pq_node k = pseudo_root->left_most;
    pq_node l = NULL;

    while (k && k != pseudo_root->right_most)
    { go_to_sibling(k, l);
      del_subtree(l); // Note that also l is deleted in this function
    call.
    // But to progress correctly in the chain we need
    l's
    // value (see previous command).
    }
}
```

```
    // loescht alle Kinder von x und dann x selber:
```

```
void pq_tree::del_subtree(pq_node x)
{
    pq_node k = x->left_most;
    pq_node l = NULL;

    while (k)
    { //Note that the value of l is essentially for the loop
      //but the pointer l is not valid
      //(see also the comment in del_pert_subtree() ).

      go_to_sibling(k, l);
      del_subtree(l); //delete the subtree recursively
    }

    delete x;
}
```

```
bool pq_tree::update(list<int>& S)
{
    int i=0;
```



```

pq_node v, w, k= NULL, l;

if (S.empty())
{
    // delete the pert. subtree contained in processed...

    while (!processed.empty())
    { w = processed.pop();
      if (w->type == D_NODE_LEAF) leaves[w->leaf_index] = NULL;
      delete w;
    }

    // if there's anything else in the PQ_tree the reduction has
failed.

    for(i=0; i < leaves_size && !leaves[i]; i++);

    return (i == leaves_size);
}

if (root && (pseudo_root->left_most == root))
{ // A special case requires special treatment.
  // Note that pseudo_root has only 1 child in this case.

  while (!processed.empty())
  { w = processed.pop();
    if (w->type == D_NODE_LEAF) leaves[w->leaf_index] = NULL;
    delete w;
  }

  // The pertinent subtree, here that means the whole pq_tree
  // is deleted. The application has failed.
  return false;
}

i = 0;
v = new pq_node_struct;

if (pseudo_root->left_most &&
    pseudo_root->left_most->parent_type == D_NODE_PNODE)
{ v->parent_type = D_NODE_PNODE;
  v->parent = pseudo_root->left_most->parent;
}
else
  if (pseudo_root->left_most) v->parent_type = D_NODE_QNODE;

if (S.size() == 1)
{
    // then create v as a
leaf
    v->leaf_index = S.pop();

    while (v->leaf_index >= leaves_size) leaves_double();

    if (leaves[v->leaf_index]) return false;

    leaves[v->leaf_index] = v;
    v->type = D_NODE_LEAF;
}
else
{ //then create v as a P-node with leaves labelled with the elements
of S
    // v->leaf_index = pseudo_root->leaf_index;

```

```

        //only for testing

v->type = D_NODE_PNODE;
v->child_count = S.size();
v->left_most = l = new pq_node_struct;
l->link_one_side = NULL;
l->parent = v;
l->leaf_index = S.pop();

while (l->leaf_index >= leaves_size) leaves_double();

if (leaves[l->leaf_index]) return false;

leaves[l->leaf_index] = l;
l->parent_type = D_NODE_PNODE;
l->type = D_NODE_LEAF;

while (!S.empty())
{ l->link_other_side = k = new pq_node_struct;
  k->leaf_index = S.pop();

  while (k->leaf_index >= leaves_size) leaves_double();

  if (leaves[k->leaf_index]) return false;

  leaves[k->leaf_index] = k;
  k->parent_type = D_NODE_PNODE;
  k->type = D_NODE_LEAF;
  k->parent = v;
  k->link_one_side = l;
  l = k;
}
k->link_other_side = NULL;
v->right_most = k;
}

if (!root)
{
    //then the pq_tree is just constructed & still
empty
    root = v;
    v->parent_type = D_NODE_QNODE;
    return true;
}

//replace the full chain under pseudo_root by
v
k = pseudo_root->left_most;
l = pseudo_root->right_most;

if (k == l) //then pseudo_root has only one child
    replace_in_siblings(k, v);
else
{
    pq_node k1, l1;

    if (k->link_one_side && k->link_other_side)
    { // Then k has a full and an empty sibling.
      // The full is contained in the pertinent subtree and to be
      // deleted. The empty becomes a sibling of v.

      if (k->link_one_side->status == D_NODE_EMPTY &&

```

```

        k->link_one_side->type != D_NODE_DIR)
        v->link_one_side = k->link_one_side;
    else
        if (k->link_other_side->status == D_NODE_EMPTY &&
            k->link_other_side->type != D_NODE_DIR)
            v->link_one_side = k->link_other_side;
        else
            if (k->link_one_side->status == D_NODE_FULL)
                v->link_one_side = k->link_other_side;
            else
                if (k->link_other_side->status == D_NODE_FULL)
                    v->link_one_side = k->link_one_side;
                else
                    { k1 = k->link_one_side; l1 = k;
                      skip_dir(k1,l1);
                      if (k1->status == D_NODE_EMPTY)
                          v->link_one_side = k->link_one_side;
                      else
                          v->link_one_side = k->link_other_side;
                    }

        if (v->link_one_side->link_one_side == k)
            v->link_one_side->link_one_side = v;
        else
            v->link_one_side->link_other_side = v;
    }
    else
    {
        // then k is an endmost of his real
father
        v->link_one_side = NULL;
        if (k->parent->right_most == k)
            k->parent->right_most = v;
        else
            k->parent->left_most = v;

        v->parent = k->parent;
    }

    // v is chained instead of k.
if (l->link_one_side && l->link_other_side)
{
    // analogous to treatment of k, as above
    if (l->link_one_side->status == D_NODE_EMPTY &&
        l->link_one_side->type != D_NODE_DIR)
        v->link_other_side = l->link_one_side;
    else
        if (l->link_other_side->status == D_NODE_EMPTY &&
            l->link_other_side->type != D_NODE_DIR)
            v->link_other_side = l->link_other_side;
        else
            if (l->link_one_side->status == D_NODE_FULL)
                v->link_other_side = l->link_other_side;
            else
                if (l->link_other_side->status == D_NODE_FULL)
                    v->link_other_side = l->link_one_side;
                else
                    { k1 = l->link_one_side; l1 = l;
                      skip_dir(k1,l1);
                      if (k1->status == D_NODE_EMPTY)
                          v->link_other_side = l->link_one_side;
                      else

```

```

        v->link_other_side = l->link_other_side;
    }
    if (v->link_other_side->link_one_side == l)
        v->link_other_side->link_one_side = v;
    else
        v->link_other_side->link_other_side = v;
}
else
{
    //then l is an endmost of his real father
    v->link_other_side = NULL;
    if (l->parent->right_most == l)
        l->parent->right_most = v;
    else
        l->parent->left_most = v;

    v->parent = l->parent;
}
// v is chained instead of l.
}

// Now v replaces the chain of full nodes beyond
// the real father.

// v is in the scanning direction of the pert subtree
// and if necessary we can add a DIR-ptr
if (pseudo_root->left_most->parent_type == D_NODE_QNODE)
{
    k = new pq_node_struct;

    k->leaf_index = pseudo_root->leaf_index;
    k->type = D_NODE_DIR; // insert a DIR-ptr, but not as an
end_most
    if (v->link_one_side)
    {
        k->link_one_side = v->link_one_side;
        v->link_one_side = k;
        k->link_other_side = v;
        k->link_one_side->link_one_side == v ?
            (k->link_one_side->link_one_side = k) :
            (k->link_one_side->link_other_side = k);
    }
    else
    {
        k->link_other_side = v->link_other_side;
        v->link_other_side = k;
        k->link_one_side = v;
        k->link_other_side->link_one_side == v ?
            (k->link_other_side->link_one_side = k) :
            (k->link_other_side->link_other_side = k);
    }
}

// Prepare pseudo_root for the next possible DIR-ptr
pseudo_root->leaf_index++;
pseudo_root->mark = D_NODE_UNMARKED;
pseudo_root->status = D_NODE_EMPTY;

// Now we delete the pertinent subtree.
while (!processed.empty())

```

```

    { w = processed.pop();
      if (w->type == D_NODE_LEAF) leaves[w->leaf_index] = NULL;
      delete w;
    }

#ifdef _DEBUG_PQ_TREE
    cons_pq_tree(root, "b", "update");
#endif

    return true;
}

// h"aufig vorkommende Aktionen:

inline void pq_tree::skip_dir(pq_node& k, pq_node& last_k)
// skip direction
indicators
{
    while (k && k->type == D_NODE_DIR)
        go_to_sibling(k, last_k);
}

inline void pq_tree::go_to_sibling(pq_node& k, pq_node& last_k)
{
    if (k->link_one_side == last_k)
    {
        last_k = k;
        k = k->link_other_side;
    }
    else
    { last_k = k;
      k = k->link_one_side;
    }
}

```

```

/*****
*****
+
+ LEDA 3.5.1
+
+ _d2_spring.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****
*****/
#include <LEDA/graph_alg.h>

#include <math.h>
#include <LEDA/array2.h>

static float log_2(int x)
{ float l = 0;
  while (x)
  { l++;
    x >>= 1;
  }
  return l/2;
}

void D2_SPRING_EMBEDDING(const graph& G, node_array<double>& xpos,
                        node_array<double>& ypos,
                        double xleft, double xright,
                        double ybottom, double ytop,
                        int iterations)
{ double width = xright - xleft;
  double height = ytop - ybottom;

  for (int count = 1; count < iterations; count++)
  {
    double k = sqrt(width*height / G.number_of_nodes()) / 2;

    //float l2 = 50*log_2(1+count);

    float l2 = 25*log_2(1+count);

    double tx = width / l2;
    double ty = height / l2;

    node_array<double> xdisp(G,0);
    node_array<double> ydisp(G,0);

    // repulsive forces

    node v;

```

```

forall_nodes(v,G)
{
    double xv = xpos[v];
    double yv = ypos[v];

    node u;
    forall_nodes(u,G)
    { if(u == v) continue;
      double xdist = xv - xpos[u];
      double ydist = yv - ypos[u];

      double dist = xdist * xdist + ydist * ydist;

      if (dist < 1e-3) dist = 1e-3;

      double frepulse = k*k/dist;

      xdisp[v] += frepulse * xdist;
      ydisp[v] += frepulse * ydist;
    }

    //xdisp[v] *= (double(rand_int(750,1250))/1000.0);
    //ydisp[v] *= (double(rand_int(750,1250))/1000.0);
}

// attractive forces

edge e;
forall_edges(e,G)
{ node u = G.source(e);
  node v = G.target(e);

  double xdist=xpos[v]-xpos[u];
  double ydist=ypos[v]-ypos[u];

  double dist=sqrt(xdist*xdist+ydist*ydist);

  float f = (G.degree(u)+G.degree(v))/16.0;

  dist /= f;

  xdisp[v]-=xdist*dist/k;
  ydisp[v]-=ydist*dist/k;
  xdisp[u]+=xdist*dist/k;
  ydisp[u]+=ydist*dist/k;
}

// preventions

forall_nodes(v,G)
{ double xd = xdisp[v];
  double yd = ydisp[v];

  double dist = sqrt(xd*xd+yd*yd);

  xd = tx*xd/dist;
  yd = ty*yd/dist;

  double xp = xpos[v] + xd;

```

```

    double yp = ypos[v] + yd;

    //if (xp > xleft && xp < xright)
        xpos[v] = xp;
    //if (yp > ybottom && yp < ytop)
        ypos[v] = yp;
    }
}

void D2_SPRING_EMBEDDING1(const graph& G, node_array<double>& xpos,
                          node_array<double>& ypos,
                          double xleft, double xright,
                          double ybottom, double ytop,
                          int iterations)
{
    double width = xright - xleft;
    double height = ytop - ybottom;

    for (int count = 1; count < iterations; count++)
    {
        double k = sqrt(width*height / G.number_of_nodes()) / 2;

        //float l2 = 50*log_2(1+count);

        float l2 = 25*log_2(1+count);

        double tx = width / l2;
        double ty = height / l2;

        node_array<double> xdisp(G,0);
        node_array<double> ydisp(G,0);

        // repulsive forces

        node v;
        forall_nodes(v,G)
        {
            double xv = xpos[v];
            double yv = ypos[v];
            node u;
            forall_nodes(u,G)
            {
                if(u == v) continue;
                double xdist = xv - xpos[u];
                double ydist = yv - ypos[u];
                double dist = xdist * xdist + ydist * ydist;
                if (dist < 1e-3) dist = 1e-3;
                double frepulse = k*k/dist;
                xdisp[v] += frepulse * xdist;
                ydisp[v] += frepulse * ydist;
            }
        }

        edge e;
        forall_edges(e,G)
        {
            node a = source(e);
            node b = target(e);
            if (a == v || b == v) continue;
            double xdist = xv - (xpos[a]+xpos[b])/2;
            double ydist = yv - (ypos[a]+ypos[b])/2;
            double dist = xdist * xdist + ydist * ydist;
            if (dist < 1e-3) dist = 1e-3;
        }
    }
}

```



```

        double frepulse = k*k/dist;
        xdisp[v] += frepulse * xdist;
        ydisp[v] += frepulse * ydist;
    }
}

// attractive forces

edge e;
forall_edges(e,G)
{ node u = G.source(e);
  node v = G.target(e);

  double xdist=xpos[v]-xpos[u];
  double ydist=ypos[v]-ypos[u];

  double dist=sqrt(xdist*xdist+ydist*ydist);

  float f = (G.degree(u)+G.degree(v))/16.0;

  dist /= f;

  xdisp[v]-=xdist*dist/k;
  ydisp[v]-=ydist*dist/k;
  xdisp[u]+=xdist*dist/k;
  ydisp[u]+=ydist*dist/k;
}

// preventions

forall_nodes(v,G)
{ double xd = xdisp[v];
  double yd = ydisp[v];

  double dist = sqrt(xd*xd+yd*yd);

  xd = tx*xd/dist;
  yd = ty*yd/dist;

  double xp = xpos[v] + xd;
  double yp = ypos[v] + yd;

  //if (xp > xleft && xp < xright)
    xpos[v] = xp;
  //if (yp > ybottom && yp < ytop)
    ypos[v] = yp;
}
}
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _d3_spring.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
+ *****/
#include <LEDA/graph_alg.h>

#include <math.h>
#include <LEDA/array2.h>

static float log_2(int x)
{ float l = 0;
  while (x)
  { l++;
    x >>= 1;
  }
  return l/2;
}

void D3_SPRING_EMBEDDING(const graph& G, node_array<double>& xpos,
                        node_array<double>& ypos,
                        node_array<double>& zpos,
                        double xleft, double xright,
                        double ybottom, double ytop,
                        double zbottom, double ztop,
                        int iterations)
{ list<node> L;
  D3_SPRING_EMBEDDING(G, L, xpos, ypos, zpos,
xleft, xright, ybottom, ytop, zbottom, ztop, iterations); }

void D3_SPRING_EMBEDDING(const graph& G, const list<node>& fixed_nodes,
                        node_array<double>& xpos,
                        node_array<double>& ypos,
                        node_array<double>& zpos,
                        double xleft, double xright,
                        double ybottom, double ytop,
                        double zbottom, double ztop,
                        int iterations)
{
  if (xleft >= xright || ybottom >= ytop || zbottom >= ztop)
    error_handler(1, "SPRING_EMBDDING: illegal bounds.");
}

```

```

double width  = xright - xleft;
double height = ytop - ybottom;
double depth  = ztop - zbottom;

for (int count = 1; count < iterations; count++)
{
    double k = sqrt(width*height / G.number_of_nodes()) / 2;

    float l2 = 50*log_2(1+count);

    double tx = width  / l2;
    double ty = height / l2;
    double tz = depth  / l2;

    node_array<double> xdisp(G,0);
    node_array<double> ydisp(G,0);
    node_array<double> zdisp(G,0);

    // repulsive forces

    node v;
    forall_nodes(v,G)
    { int i = int((xpos[v] - xleft) / k);
      int j = int((ypos[v] - ybottom) / k);

      double xv = xpos[v];
      double yv = ypos[v];
      double zv = zpos[v];

      node u;
      forall_nodes(u,G)
      { if(u==v) continue;
        double xdist = xv - xpos[u];
        double ydist = yv - ypos[u];
        double zdist = zv - zpos[u];

        double dist = xdist * xdist + ydist * ydist + zdist * zdist;

        if (dist < 1e-3) dist = 1e-3;

        double frepulse = k*k/dist;

        xdisp[v] += frepulse * xdist;
        ydisp[v] += frepulse * ydist;
        zdisp[v] += frepulse * zdist;
      }
      //xdisp[v] *= (double(rand_int(750,1250))/1000.0);
      //ydisp[v] *= (double(rand_int(750,1250))/1000.0);
      //zdisp[v] *= (double(rand_int(750,1250))/1000.0);
    }

    // attractive forces

    edge e;
    forall_edges(e,G)
    { node u = G.source(e);
      node v = G.target(e);

      double xdist=xpos[v]-xpos[u];

```

```

double ydist=ypos[v]-ypos[u];
double zdist=zpos[v]-zpos[u];

double dist=sqrt(xdist*xdist+ydist*ydist+zdist*zdist);

float f = (G.degree(u)+G.degree(v))/16.0;

dist /= f;

xdisp[v]-=xdist*dist/k;
ydisp[v]-=ydist*dist/k;
zdisp[v]-=zdist*dist/k;
xdisp[u]+=xdist*dist/k;
ydisp[u]+=ydist*dist/k;
zdisp[u]+=zdist*dist/k;
}

// preventions
forall_nodes(v,G)
{ double xd = xdisp[v];
  double yd = ydisp[v];
  double zd = zdisp[v];

  double dist = sqrt(xd*xd+yd*yd+zd*zd);

  xd = tx*xd/dist;
  yd = ty*yd/dist;
  zd = tz*zd/dist;

  double xp = xpos[v] + xd;
  double yp = ypos[v] + yd;
  double zp = zpos[v] + zd;

  if (xp > xleft && xp < xright) xpos[v] = xp;
  if (yp > ybottom && yp < ytop) ypos[v] = yp;
  if (zp > zbottom && zp < ztop) zpos[v] = zp;
}
}
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _embed1.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****/

//-----
//
// straight line embedding
//
// K. Mehlhorn (1989)
//-----

#include <LEDA/graph_alg.h>

const int A = -2;
const int B = -1;

static node_array<list_item> Classloc;
static node_array<int> ord, labelled, Class;
static node_array<node> first, second, last;

void label_node(graph& G, list<node>& L, int& count,
               list<node>& A1, list<node>& B1, list<node>*& I1,
               node v, node c)

{ // labels the node v; c is the special node which is to be labelled
  // last; the details are described in lemma 10
  edge e;
  L.append(v);
  ord[v]=count++;
  labelled[v]=1;

  forall_adj_edges(e,v)
  { edge e1 = G.reversal(e);
    node tt = target(e);
    int i;

    if (labelled[tt] && !labelled[target(G.cyclic_adj_succ(e))])
    { first[v]=tt;
      second[v]=target(G.cyclic_adj_pred(e));
    }
  }
}

```



```

/*
  node_array<int> Class(G);
  node_array<list_item> Classloc(G);
*/

Class.init(G);
Classloc.init(G);

forall_nodes(v,G) { Classloc[v] = A1.push(v);Class[v]=A;}

list<node> B1;
list<node>* I1 = new list<node>[G.number_of_nodes()];

label_node(G,L,count,A1,B1,I1,a,c);
label_node(G,L,count,A1,B1,I1,b,c);

while ( !I1[1].empty() )
{ node v = I1[1].pop();
  label_node(G,L,count,A1,B1,I1,v,c);
}

label_node(G,L,count,A1,B1,I1,c,c);

//nun berechne ich noch first second und last des Knoten c
first[c]=a;
last[c]=b;

edge e;
forall_adj_edges(e,c) if (target(e)==a)
second[c]=target(G.cyclic_adj_pred(e));

//nun die Folge Pi
node_array<list_item> Piloc(G);
Piloc[a] = Pi.push(a);
Piloc[b] = Pi.append(b);
forall(v,L) if (v != a && v != b) Piloc[v] = .
Pi.insert(v,Piloc[second[v]],-1);

} //end of compute_labelling

void move_to_the_right(list<node>& Pi, node v, node w,
                      node_array<int>& ord, node_array<int>& x)
{ //increases the x-coordinate of all nodes which follow w in List Pi
  //and precede v in List L,i.e., have a smaller ord value than v
  int seen_w = 0;
  node z;
  forall(z,Pi)
  { if (z==w) seen_w=1;
    if (seen_w && (ord[z]<ord[v])) x[z]=x[z]+1;
  }
}

int STRAIGHT_LINE_EMBEDDING(graph& G,node_array<int>& x,
node_array<int>& y)
{
  // computes a straight-line embedding of the planar map G into
  // the 2n by n grid. The coordinates of the nodes are returned
  // in the Arrays x and y. Returns the maximal coordinate.

```

```

if (G.empty()) return 0;

if (G.number_of_nodes() == 1)
{ node v = G.first_node();
  x[v] = y[v] = 1;
  return 1;
}

list<node> L;
list<node> Pi;
list<edge> TL;

node v;
edge e;
int maxcoord = 1;

/*
node_array<int> ord(G);
node_array<node> first(G), second(G), last(G);
*/

ord.init(G);
first.init(G);
second.init(G);
last.init(G);

TL = G.triangulate_map();

if (!G.make_map())
  error_handler(1, "STRAIGHT LINE EMBEDDING: graph must be a planar
map");

compute_labelling(G, L, Pi);

//I now embed the first three nodes

v = L.pop();
x[v] = 0;
y[v] = 0;

if (!L.empty())
{ v = L.pop();
  x[v] = 2;
  y[v] = 0;
}

if (!L.empty())
{ v = L.pop();
  x[v] = 1;
  y[v] = 1;
}

//I now embed the remaining nodes

while ( !L.empty() )
{
  v = L.pop();

```



```

// I first move the nodes depending on second[v] by one unit
// and the the nodes depending on last[v] by another unit to the
// right

move_to_the_right(Pi,v,second[v],ord,x);
move_to_the_right(Pi,v,last[v],ord,x);

// I now embed v at the intersection of the line with slope +1
// through first[v] and the line with slope -1 through last[v]

int x_first_v = x[first[v]];
int x_last_v = x[last[v]];
int y_first_v = y[first[v]];
int y_last_v = y[last[v]];

x[v]=(y_last_v - y_first_v + x_first_v + x_last_v)/2;
y[v]=(x_last_v - x_first_v + y_first_v + y_last_v)/2;
}

// delete triangulation edges
forall(e,TL) G.del_edge(e);

forall_nodes(v,G) maxcoord = Max(maxcoord,Max(x[v],y[v]));

return maxcoord;
}

void STRAIGHT_LINE_EMBEDDING(graph& G,node_array<double>& x,
node_array<double>& y)
{
node_array<int> x0(G);
node_array<int> y0(G);

int maxc = STRAIGHT_LINE_EMBEDDING(G,x0,y0);

node v;
forall_nodes(v,G)
{ x[v] = double(x0[v])/maxc;
  y[v] = double(y0[v])/maxc;
}
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _embed2.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
+ *****/

//-----
//
// Dirk Ambras 1995
//-----

#include <LEDA/graph_alg.h>

static void Contract(graph& G, node a, node b , node c, list<node>& L)
{
    node          v,w;
    list<node>     cand;
    node_array<bool> marked(G,false);           // betrachtete Knoten
    node_array<int> deg(G,0);                   // # virtuelle Nachbarn

    int N = G.number_of_edges();

    marked[a] = marked[b] = marked[c] = true;           // Init

    deg[a] = deg[b] = deg[c] = N;

    forall_adj_nodes(v,a)
    { marked[v]=true;
      forall_adj_nodes(w,v) deg[w]++;                // mache v bekannt bei den w's
    }

    { forall_adj_nodes(v,a)                          // lade Kandidaten
      if (deg[v]<=2) cand.append(v);
    }

    while (!cand.empty())
    { node u=cand.pop();
      if (deg[u] == 2)
      { L.push(u);
        deg[u]=N;
        forall_adj_nodes(v, u)
        { deg[v]--;
          if (!marked[v])                // u ist virtuell geloesch
            { marked[v]=true;           // v ist neuer Nachbar von a
        }
      }
    }
}

```

```

        forall_adj_nodes(w,v) deg[w]++;      // mache v bekannt bei
den w's
        if (deg[v] <= 2) cand.append(v);    // lade Kandidaten
        } else
        if (deg[v] == 2) cand.append(v);
    }
}
}

```

```

static void Realizer(graph& G, const list<node>& L,
                    node a, node b, node c,
                    GRAPH<node, int>& T, node_array<node>& v_in_T)
{
    int i=0;
    node v;
    edge e;
    node_array<int> ord(G,0);

    ord[b] = i++;
    ord[c] = i++;
    node u;
    forall(u,L) ord[u]=i++;                // V(G) numerieren
    ord[a] = i++;

    forall_nodes(v, G) v_in_T[v] = T.new_node();    // T = copy of G

    forall(v, L)
    { node u = v_in_T[v];    // u is copy of v in T

        forall_adj_edges(e, v)
        if (ord[G.target(e)] > ord[v]) break;

        edge e1 = e;
        while(ord[G.target(e1)] > ord[v]) e1 = G.cyclic_adj_succ(e1);
        T.new_edge(v_in_T[G.target(e1)], u, 2);

        edge e2 = e;
        while(ord[G.target(e2)] > ord[v]) e2 = G.cyclic_adj_pred(e2);
        T.new_edge(v_in_T[G.target(e2)], u, 3);

        for(e=G.cyclic_adj_succ(e1); e != e2; e=G.cyclic_adj_succ(e))
            T.new_edge(u, v_in_T[G.target(e)], 1);
    }

    // special treatement of a,b,c

    node a_in_T = v_in_T[a];
    node b_in_T = v_in_T[b];
    node c_in_T = v_in_T[c];

    forall_adj_edges(e,a)
    T.new_edge(a_in_T, v_in_T[G.target(e)], 1);

    T.new_edge(b_in_T, a_in_T, 2);
    T.new_edge(b_in_T, c_in_T, 2);

    T.new_edge(c_in_T, a_in_T, 3);
    T.new_edge(c_in_T, b_in_T, 3);
}

```

```

}

static void Subtree_Sizes(GRAPH<node, int>& T, int i, node r,
                        node_array<int>& size)
{
    // computes sizes of all subtrees of tree with root r in T(i)

    int sum=0;
    edge e;
    forall_adj_edges(e, r)
        if (T[e]==i)
        { node w=T.target(e);
          Subtree_Sizes(T, i, w, size);
          sum+=size[w];
        }
    size[r]=sum+1;
}

static void Prefix_Sum(GRAPH<node, int>& T, int i, node r,
                      const node_array<int>& val, node_array<int>& sum)
{
    // computes for every node u in the subtree of T(i) with root r
    // the sum of all val[v] where v is a node on the path from r to u

    list<node> Q;

    Q.append(r);
    sum[r] = val[r];

    while (!Q.empty())
    { node v=Q.pop();
      edge e;
      forall_adj_edges(e, v)
          if (T[e]==i)
          { node w=T.target(e);
            Q.append(w);
            sum[w] = val[w] + sum[v];
          }
    }
}

int STRAIGHT_LINE_EMBEDDING2(graph& G, node_array<int>& xcoord,
                             node_array<int>& ycoord)
{
    int n = G.number_of_nodes();

    if (n < 3)
    { int max_c = 1;
      if (n > 0)
      { node a = G.first_node();
        xcoord[a] = 1;
        ycoord[a] = 1;
      }
      if (n > 1)
      { node b = G.last_node();
        xcoord[b] = 2;
        ycoord[b] = 2;
      }
    }
}

```

```

    max_c = 2;
}
return max_c;
}

node          v;
list<node>     L;
GRAPH<node, int> T;
node_array<node> v_in_T(G);

list<edge> el = G.triangulate_map();

// choose outer face a,b,c

node a=G.first_node();
edge e=G.first_adj_edge(a);
node c=G.target(e);
node b = G.target(G.adj_succ(e));

Contract(G,a,b,c,L);

Realizer(G,L,a,b,c,T,v_in_T);           // T aufbauen

node_array<int>  t1(T);
node_array<int>  t2(T);
node_array<int>  val(T,1);

node_array<int>  P1(T);
node_array<int>  P3(T);
node_array<int>  v1(T);
node_array<int>  v2(T);

Subtree_Sizes(T, 1, v_in_T[a], t1);
Subtree_Sizes(T, 2, v_in_T[b], t2);

Prefix_Sum(T, 1, v_in_T[a], val, P1);
Prefix_Sum(T, 3, v_in_T[c], val, P3);
// now Pi = depth of all nodes in Tree T(i) (depth[root] = 1)

Prefix_Sum(T, 2, v_in_T[b], t1, v1);
v1[v_in_T[a]] = t1[v_in_T[a]]; // Sonderrolle von a

// in v1[v] steht jetzt die Summe (Anzahl der Knoten im T1-UBaum[x])
// fuer jeden Knoten x im Pfad in T2 von b nach v

Prefix_Sum(T, 3, v_in_T[c], t1, val);
val[v_in_T[a]] = t1[v_in_T[a]]; // Sonderrolle von
a

// in val[v] steht jetzt die Summe (Anzahl der Knoten im T1-UBaum[x])
// fuer jeden Knoten x im Pfad in T3 von c nach v
// es ist r1[v]=v1[v]+val[v]-t1[v] die Anzahl der Knoten in der
// Region 1 von v

forall_nodes(v, T) v1[v] += val[v]-t1[v]-P3[v]; // v1' errechnen

Prefix_Sum(T, 3, v_in_T[c], t2, v2);

```

```

    v2[v_in_T[b]]=t2[v_in_T[b]]; // Sonderrolle von
b
    Prefix_Sum(T, 1, v_in_T[a], t2, val);
    val[v_in_T[b]]=t2[v_in_T[b]]; // Sonderrolle von
b
    forall_nodes(v, T) v2[v] += val[v]-t2[v]-P1[v]; // v2' errechnen

    int maxcoord = 0;

    forall_nodes(v, G) // x- & y-Feld kopieren
    { xcoord[v] = v1[v_in_T[v]];
      ycoord[v] = v2[v_in_T[v]];
      maxcoord = Max(maxcoord,Max(xcoord[v],ycoord[v]));
    }

    forall(e, el) G.del_edge(e); //
eingefuegte Kanten // loeschen

    return maxcoord;
}

void STRAIGHT_LINE_EMBEDDING2(graph& G,node_array<double>& x,
node_array<double>& y)
{
    node_array<int> x0(G);
    node_array<int> y0(G);

    int maxc = STRAIGHT_LINE_EMBEDDING2(G,x0,y0);

    node v;
    forall_nodes(v,G)
    { x[v] = double(x0[v])/maxc;
      y[v] = double(y0[v])/maxc;
    }
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _ortho.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****/

#include <LEDA/planar_map.h>
#include <LEDA/node_map.h>
#include <LEDA/edge_map.h>
#include <LEDA/face_map.h>
#include <LEDA/stack.h>
#include <LEDA/graph_alg.h>

#define EPS ""
#define INFINITY MAXINT
#define REV(e) P.reversal(e)
#define SUCC(e) P.face_cycle_succ(e)
#define PRED(e) P.face_cycle_pred(e)
#define IS_CAGE(f) (P.get_cage(P.first_face_edge(f)))
#define IS_OUTER(f) (!P.get_inner(P.first_face_edge(f)))
#define NEXT(d) ((direction)((d+5)%4))
#define PREV(d) ((direction)((d+3)%4))
#define OPP(d) ((direction)((d+6)%4))

#define ERR_EMPTY_GRAPH "ORTHO: input graph is empty"
#define ERR_NOT_CONNECTED "ORTHO: input graph is not connected"
#define ERR_NO_PLANAR_MAP "ORTHO: this is no planar map"
#define ERR_INVALID_NETWORK "ORTHO: invalid network"
#define ERR_NO_FEASIBLE_FLOW "ORTHO: no feasible flow"
#define ERR_NO_ORTHO_REP "ORTHO: orthogonal rep. not valid"
#define ERR_BAD_STRING "ORTHO: trying to set nonempty string"
#define ERR_BAD_ANGLE "ORTHO: bad angle"
#define ERR_BAD_DIRECTION "ORTHO: bad direction"
#define ERR_OPEN_CPLEX_ENV "CPLEX: failed to open environment"
#define ERR_LOAD_LP "CPLEX: failed to load LP"
#define ERR_CPLEX_ADD_ROWS "CPLEX: CPXaddrows failed"
#define ERR_CPLEX_WRITE "CPLEX: CPXwriteLP failed"
#define ERR_CPLEX_OPT "CPLEX: failed to optimize LP"
#define ERR_CPLEX_GET_SOLUTION "CPLEX: failed to obtain solution"
#define ERR_CPLEX_FREE "CPLEX: CPXfreeprob failed"
#define ERR_CPLEX_CLOSE "CPLEX: CPXcloseCPLEX failed"

enum direction{north,east,south,west,unexplored};
enum v_type{real,bend,dissection,big};

void longest_paths(const GRAPH<int,int>&G,node_array<int>&l){

```

```

node_array<int>INDEG(G,0);node v;edge e;
stack<node>S;
forall_nodes(v,G){
INDEG[v]= indeg(v);
if(INDEG[v]==0)S.push(v);
}
while(!S.empty()){
v= S.pop();
forall_out_edges(e,v){
node w= target(e);
l[w]= Max(l[w],l[v]+G.inf(e));
if(--INDEG[w]==0)S.push(w);
}
}
}

```

```

int angle(const char c){
int result= 0;
switch(c){
case'0':result= 90;break;
case'1':result= 270;break;
default:error_handler(1,ERR_BAD_ANGLE);
}
return result;
}

```

```

bool Euler(const graph&G){
int n= 0;node v;forall_nodes(v,G)if(outdeg(v)!=0)++n;
int m= G.number_of_edges()/2;
int f= G.number_of_faces();
return(m==n+f-2);
}

```

```

class ortho_map:public planar_map{
node_map<node>v_in_G;
node_map<v_type>v_T;
edge_map<edge>e_in_G;
edge_map<edge>e_in_P;
edge_map<int>a;
edge_map<string>s;
edge_map<bool>inner;
edge_map<bool>cage;
edge_array<direction>dir;
edge_array<int>length;
node_array<int>x,y;

public:
edge next_level_edge(const node,const direction);
ortho_map(const graph&);
void print();
//bool check();
void init_maps(const edge,const int,const string,const bool,const bool);
node get_orig(const node v)const{return v_in_G[v];}
edge get_orig(const edge e)const{return e_in_G[e];}

```



```

direction get_dir(const edge e)const{return dir[e];}
edge get_copy(const edge e)const{return e_in_P[e];}
v_type get_type(const node v)const{return v_T[v];}
void set_inner(const edge e,const bool t){inner[e]= t;}
void set_cage(const edge e,const bool t){cage[e]= t;}
bool get_inner(const edge e)const{return inner[e];}
bool get_cage(const edge e)const{return cage[e];}
void set_type(const node v,const v_type t){v_T[v]= t;}
int get_a(const edge e)const{return a[e];}
void set_a(const edge e,const int angle){a[e]= angle;}
string get_s(const edge e)const{return s[e];}
void set_s(const edge e,const string s_new){s[e]= s_new;}
void set_s(const edge e,const int,const int,const bool);
int get_x(const node v)const{return x[v];}
int get_y(const node v)const{return y[v];}
void set_rev_s(const edge e,const int,const int,const bool);
void set_length(const edge e,const int l_new){length[e]= l_new;}
int get_length(const edge e)const{return length[e];}
edge split_map_edge(edge);
edge new_edge(edge,edge);
edge split_bend_edge(edge);
void PrintEdge(const edge);
void init_rest(){dir.init(*this,unexplored);
length.init(*this,0);
x.init(*this,-1);y.init(*this,-1);}

```

```

void ortho_map::assign_directions(edge e,direction d){
while(dir[e]==unexplored){
dir[e]= d;edge r= reversal(e);
if(dir[r]==unexplored){
if(inner[r])assign_directions(r,OPP(d));
else dir[r]= OPP(d);
}
switch(a[e]){
case 90:d= NEXT(d);break;
case 270:d= PREV(d);break;
case 360:d= OPP(d);break;
}
e= face_cycle_succ(e);
}
}

```

```

void determine_position(const node,const int x,const int
y,node_array<bool>&);
void norm_positions();
edge succ_corner_edge(const edge);
~ortho_map(){clear();}
LEDA_MEMORY(ortho_map)
};

```

```

edge ortho_map::next_level_edge(const node v,const direction d){
edge e;
switch(d){
case east: { forall_out_edges(e,v) if(dir[e]==north) return e; break; }
case west: { forall_in_edges(e,v) if(dir[e]==north) return e; break; }
case north:{ forall_in_edges(e,v) if(dir[e]==east) return e; break; }

```

```

case south:{ forall_out_edges(e,v) if(dir[e]==east) return e; break; }
default:break;
}
return NULL;
}

```

```

ortho_map::ortho_map(const graph&G):planar_map(G){
if(G.number_of_nodes()==0)error_handler(1,ERR_EMPTY_GRAPH);
if(!Is_Connected(G))error_handler(1,ERR_NOT_CONNECTED);
if(!Euler(G))error_handler(1,ERR_NO_PLANAR_MAP);
v_in_G.init(*this);e_in_G.init(*this);
e_in_P.init(G);

```

```

node v_P= first_node(),v_G= G.first_node();
while(v_P){
v_in_G[v_P]= v_G;
v_P= succ_node(v_P);
v_G= G.succ_node(v_G);
}
forall_nodes(v_P,*this){
v_G= v_in_G[v_P];
edge e_P= first_adj_edge(v_P),e_G= G.first_adj_edge(v_G);
while(e_P){
e_in_G[e_P]= e_G;e_in_P[e_G]= e_P;
e_G= adj_succ(e_G);e_P= adj_succ(e_P);
}
}

```

```

;
a.init(*this,90);s.init(*this,EPS);v_T.init(*this,real);
inner.init(*this,true);cage.init(*this,false);
}

```

```

edge ortho_map::split_map_edge(edge e){
edge n= planar_map::split_edge(e);
edge er= reversal(e),nr= reversal(n);
a[er]= a[nr];a[n]= a[e];
s[er]= EPS;s[n]= EPS;
inner[er]= inner[nr];cage[er]= cage[nr];
inner[n]= inner[e];cage[n]= cage[e];
e_in_G[n]= e_in_G[e];
e_in_G[er]= e_in_G[nr];
v_in_G[source(n)]= NULL;

return n;
}

```

```

edge ortho_map::split_bend_edge(edge e){
string s_e= s[e];int a_e= a[e];
bool inner_e= inner[e];bool cage_e= cage[e];
edge er= reversal(e);
string s_er= s[er];int a_er= a[er];

```

```

bool inner_er= inner[er];bool cage_er= cage[er];

edge n= split_map_edge(e);
er= reversal(e);edge nr= reversal(n);

a[e]= angle(s_e[s_e.length()-1]);a[er]= a_er;
a[n]= a_e;a[nr]= angle(s_er[0]);
s[e]= s_e.head(s_e.length()-1);s[er]= s_er.tail(s_er.length()-1);
s[n]= EPS;s[nr]= EPS;
inner[e]= inner[n]= inner_e;
cage[e]= cage[n]= cage_e;
inner[er]= inner[nr]= inner_er;
cage[er]= cage[nr]= cage_er;
v_T[source(n)]= bend;

return n;
}

edge ortho_map::new_edge(edge e1,edge e2){
edge n= planar_map::new_edge(e1,e2);
e_in_G[n]= NULL;e_in_G[reversal(n)]= NULL;
return n;
}

void ortho_map::print(){
face f;edge e;
forall_faces(f,(*this)){
if(!inner[first_face_edge(f)])cout<<"outer ";
if(cage[first_face_edge(f)])cout<<"cage ";
cout<<"face: ";<<endl;
forall_face_edges(e,f){PrintEdge(e);newline;}
}
}

/*
bool ortho_map::check(){
bool result= true;return result;
face f;edge e;node v;
forall_faces(f,(*this)){
int rotation= 0;
forall_face_edges(e,f)rotation+= zeroes(s[e])-ones(s[e])+2-a[e]/90;
if(!inner[first_face_edge(f)])result= result&&(rotation==4);
else result= result&&(rotation==4);
}
forall_nodes(v,(*this))if(outdeg(v)!=0){
int angle_sum= 0;
forall_in_edges(e,v)angle_sum+= a[e];
result= result&&(angle_sum==360);
}
return result;
}
*/

void ortho_map::set_s(const edge e,const int flow,const int flow_rev,
const bool bridge){
if(s[e]!=EPS)error_handler(1,ERR_BAD_STRING);
for(int i= 0;i<flow;i++)s[e]+= "0";
if(!bridge){
for(int i= 0;i<flow_rev;i++)s[e]+= "1";
}
}

```

```

}

void ortho_map::set_rev_s(const edge e,const int flow,const int
flow_rev,
const bool bridge){
if(s[e]!=EPS)error_handler(1,ERR_BAD_STRING);
for(int i= 0;i<flow;i++)s[e]+= "1";
if(!bridge){
for(int i= 0;i<flow_rev;i++)s[e]= string('0')+s[e];
}
}

void ortho_map::PrintEdge(const edge e){
planar_map::print_edge(e);
cout<<"\t("<<s[e]<<" "<<a[e]<<" ["<<inner[e]<<cage[e]<<"]");
}

void ortho_map::init_maps(const edge e,const int a_new,const string
s_new,
const bool i_new,const bool c_new){
a[e]= a_new;s[e]= s_new;
inner[e]= i_new;cage[e]= c_new;
}

void ortho_map::determine_position(const node v,const int x_new,
const int y_new,node_array<bool>&seen){
if(seen[v])return;
edge e;x[v]= x_new;y[v]= y_new;
seen[v]= true;
forall_out_edges(e,v){
if(dir[e]==north)
determine_position(target(e),x_new+length[e],y_new,seen);
if(dir[e]==west)
determine_position(target(e),x_new,y_new+length[e],seen);
}
forall_in_edges(e,v){
if(dir[e]==north)
determine_position(source(e),x_new-length[e],y_new,seen);
if(dir[e]==west)
determine_position(source(e),x_new,y_new-length[e],seen);
}
}

void ortho_map::norm_positions(){
int xmin= 0,ymin= 0;node v;
forall_nodes(v,(*this))if(v_T[v]!=big){
xmin= Min(xmin,x[v]);
ymin= Min(ymin,y[v]);
}
forall_nodes(v,(*this)){
x[v]-= xmin;
y[v]-= ymin;
}
}

edge ortho_map::succ_corner_edge(const edge e){
edge e_c;
for(e_c= face_cycle_succ(e);a[e_c]==180;e_c= face_cycle_succ(e_c));
return e_c;
}

```

```

/*
bool Euler(const graph&);
int angle(const char);
int zeroes(const string&);
int ones(const string&);
void longest_paths(const GRAPH<int,int>&,node_array<int>&);
*/

/*
#include "common.h"
#include "ortho_map.h"
*/

#include <LEDA/stack.h>
#include <LEDA/array.h>
#include <LEDA/set.h>
#include <LEDA/integer_matrix.h>
#include <LEDA/graph_alg.h>

#ifdef CPLEX
extern"C"
#include<cplex.h>
#endif

typedef list<int> intlist;
typedef list<node> nodelist;

int ORTHO_EMBEDDING(const graph&G,
                    node_array<int>&x_pos,
                    node_array<int>&y_pos,
                    edge_array<intlist>&x_bends,
                    edge_array<intlist>&y_bends, bool )
{
ortho_map P(G);

node_map<edge>corr_cage_edge(P);
edge_array<nodelist>b_nodes(G);
edge_array<node>b_nodes_first(G,NULL);
edge_array<node>b_nodes_last(G,NULL);
list<node>all_nodes= P.all_nodes();
node v;edge e;face f;
forall(v,all_nodes)if(outdeg(v)>4){
P.set_type(v,big);
int d= outdeg(v),i= 0;
array<edge>out(d);edge e_cage;

```

```

forall_out_edges(e,v){
    out[i++]= e; edge e_orig= P.get_orig(e);
    edge e_split= P.split_map_edge(e);
    node c_i= source(e_split);
    P.set_type(c_i,bend);
    b_nodes_first[e_orig]= c_i;
    b_nodes_last[G.reversal(e_orig)]= c_i;
}

for(i= 0;i<d;i++){
    e_cage= P.new_edge(SUCC(out[i]),REV(out[(i+1)%d]));
    P.init_maps(e_cage,90,EPS,true,true);
    edge r_cage= P.reversal(e_cage);
    P.init_maps(r_cage,90,EPS,true,false);
}

corr_cage_edge[v]= e_cage;
forall_out_edges(e,v)P.join_faces(e);

}
//cout<<"cages created"<<endl;

//if(check)if(!Euler(P))error_handler(1,ERR_NO_PLANAR_MAP);

P.compute_faces();
face f_0;int f_0_deg= 0;
forall_faces(f,P)
if(!IS_CAGE(f)&&P.size(f)>f_0_deg){
    f_0= f;f_0_deg= P.size(f_0);
}
forall_face_edges(e,f_0)P.set_inner(e,false);

;

graph N;
node s,t;
edge_array<int>cap,cost,l;
int z= 0;

list<node>V_hat;
list<node>F;
node_map<node>NtoV(N);
node_map<face>NtoF(N);

```

```

face_map<node>FtoN(P);

;

s= N.new_node();
t= N.new_node();
forall_nodes(v,P) if (outdeg(v)>0&&outdeg(v)<=3) {
node n= N.new_node();
NtoV[n]= v;
V_hat.append(n);
}
forall_faces(f,P) {
node n= N.new_node();
NtoF[n]= f;FtoN[f]= n;
F.append(n);
}

;

list<edge>A_s_v,A_s_f,A_v,A_v_cage,A_f,A_f_t;

forall_faces(f,P) {
int size= P.size(f);
if(!IS_OUTER(f)&&size<=3)
A_s_f.append(N.new_edge(s,FtoN[f]));
if(!IS_OUTER(f)||size>=5)A_f_t.append(N.new_edge(FtoN[f],t));
}
forall(v,V_hat)A_s_v.append(N.new_edge(s,v));

;

forall(v,V_hat){
set<face>F_adj;
forall_adj_faces(f,NtoV[v])F_adj.insert(f);
forall(f,F_adj)
if(IS_CAGE(f))A_v_cage.append(N.new_edge(v,FtoN[f]));
else A_v.append(N.new_edge(v,FtoN[f]));
}

;

#define MAX_BENDS_PER_EDGE INFINITY
edge_array<bool>marked(P,false);
edge_map<edge>partner(N),a_in_P(N);edge a1,a2;
forall_faces(f,P)
forall_face_edges(e,f) if(!marked[e]){
face g= P.face_of(REV(e));
if(f==g){
a1= N.new_edge(FtoN[f],FtoN[f]);
A_f.append(a1);
partner[a1]= a1;a_in_P[a1]= e;
}else{

```

```

a1= N.new_edge(FtoN[f],FtoN[g]);
a2= N.new_edge(FtoN[g],FtoN[f]);
A_f.append(a1);A_f.append(a2);
partner[a1]= a2;partner[a2]= a1;
a_in_P[a1]= e;a_in_P[a2]= e;
edge e_loop;
for(e_loop= e;
P.face_of(REV(e_loop))==g&&!marked[e_loop];
e_loop= SUCC(e_loop))
marked[e_loop]= marked[REV(e_loop)]= true;
for(e_loop= PRED(e);
P.face_of(REV(e_loop))==g&&!marked[e_loop];
e_loop= PRED(e_loop))
marked[e_loop]= marked[REV(e_loop)]= true;
}
}

;

;

cap.init(N,0);cost.init(N,0);int z_s= 0;edge a;
l.init(N,0);
forall(a,A_s_f)cap[a]= 4-P.size(NtoF[target(a)]);
forall(a,A_s_v)cap[a]= 4-outdeg(NtoV[target(a)]);
forall(a,A_v)cap[a]= 3;
forall(a,A_v_cage){cap[a]= 3;l[a]= 1;}
forall(a,A_f){
face g= NtoF[target(a)];face f= NtoF[source(a)];
cap[a]= IS_CAGE(g)?0:MAX_BENDS_PER_EDGE;
cost[a]= IS_CAGE(f)?0:1;
}
forall(a,A_f_t){
face f= NtoF[source(a)];
if(IS_OUTER(f))cap[a]= P.size(f)+4;
else cap[a]= P.size(f)-4;
}

forall_out_edges(a,s)z_s+= cap[a];
forall_in_edges(a,t)z+= cap[a];

A_v.conc(A_v_cage);

if(z!=z_s)error_handler(1,ERR_INVALID_NETWORK);

;
//cout<<"network constructed"<<endl;

edge_array<int>flow(N);
node_array<int>supply(N,0);
supply[s]= z;supply[t]= -z;
bool feasible= MIN_COST_FLOW(N,l,cap,cost,supply,flow);

```



```

if(!feasible)error_handler(1,ERR_NO_FEASIBLE_FLOW);
//cout<<"min-cost flow computed"<<endl;

```

```

forall(a,A_v){
node v= NtoV[source(a)];face f= NtoF[target(a)];
edge e_in;bool flag= false;
forall_in_edges(e_in,v)
if(P.face_of(e_in)==f&&!flag){
P.set_a(e_in,(flow[a]+1)*90);
flag= true;
}
}

```

```

marked.init(N,false);int no_of_bends= 0;
forall(a,A_f)if(!marked[a]){
edge a_rev= partner[a];
marked[a]= true;marked[a_rev]= true;
e= a_in_P[a];
bool bridge= (a==a_rev);
P.set_s(e,flow[a],flow[a_rev],bridge);
P.set_rev_s(REV(e),flow[a],flow[a_rev],bridge);
no_of_bends+= (flow[a]+flow[a_rev]);
}

```

```

marked.init(P,false);
list<edge>all_edges= P.all_edges();
forall(e,all_edges)if(!marked[e]){
marked[e]= true;marked[REV(e)]= true;
edge e_orig= P.get_orig(e);
while(P.get_s(e)!=EPS){
edge e_split= P.split_bend_edge(e);
if(e_orig){
b_nodes[e_orig].push(source(e_split));
b_nodes[G.reversal(e_orig)].append(source(e_split));
}
}
if(e_orig){
if(b_nodes_first[e_orig]){
b_nodes[e_orig].push(b_nodes_first[e_orig]);
b_nodes[G.reversal(e_orig)].append(b_nodes_first[e_orig]);
}
if(b_nodes_last[e_orig]){
b_nodes[e_orig].append(b_nodes_last[e_orig]);
b_nodes[G.reversal(e_orig)].push(b_nodes_last[e_orig]);
}
}
}

```

```
}
```

```
//cout<<"orthogonal representation constructed"<<endl;  
//if(check)if(!P.check())error_handler(1,ERR_NO_ORTHO_REP);  
//cout<<"no. of bends in 4-graph: "<<no_of_bends<<endl;
```

```
list<face>all_faces;
```

```
forall_faces(f,P)if(IS_OUTER(f))all_faces.push(f);  
else{  
  if(IS_CAGE(f)){  
    int pred_angle= P.get_a(PRED(P.first_face_edge(f)));  
    forall_face_edges(e,f){  
      int act_angle= P.get_a(e);  
      if(pred_angle==90&&act_angle==90){  
        edge e_split= P.split_map_edge(e);  
        P.set_a(e,180);P.set_a(REV(e_split),180);  
        P.set_type(source(e_split),dissection);  
      }  
      pred_angle= P.get_a(e);  
    }  
  }  
  all_faces.append(f);  
}
```

```
forall(f,all_faces){  
  stack<edge>S;  
  int size= 0;  
  forall_face_edges(e,f)size+= abs(2-P.get_a(e)/90);  
  e= P.succ_corner_edge(P.first_face_edge(f));int state= 0;  
  while(size>4){  
    if(state==0&&!S.empty()){  
      e= P.succ_corner_edge(S.top());  
      state= 1;  
    }  
    int angle= P.get_a(e);  
    switch(angle){  
      case 90:if(state==2)state= 3;  
              if(state==1)state= 2;  
              break;  
      case 270:  
      case 360:if(state==2&&IS_OUTER(f))state= 4;  
              else{  
                S.push(e);state= 1;  
                if(angle==360)S.push(e);  
              }  
              break;  
    }  
    if(state==3){
```

```

edge e1= S.pop();
edge e2= P.succ_corner_edge(e1);
edge e3= P.succ_corner_edge(e2);
edge e4= P.succ_corner_edge(e3);int a4= P.get_a(e4);
edge e5= P.split_map_edge(e4);
P.set_type(source(e5),dissection);
P.set_a(e1,P.get_a(e1)-90);
P.set_a(e5,a4);P.set_a(REV(e5),180);
P.set_a(e4,90);
edge e6= P.new_edge(P.face_cycle_succ(e1),e5);
P.init_maps(e6,90,EPS,P.get_inner(e1),false);
P.init_maps(REV(e6),90,EPS,true,false);
P.set_inner(e2,true);P.set_inner(e3,true);P.set_inner(e4,true);
size-= 2;state= 0;e= e6;

}
if(state==4){

edge e1= S.pop();
edge e2= P.succ_corner_edge(e1);
edge e3= P.succ_corner_edge(e2);
edge e4= P.new_edge(P.face_cycle_succ(e1),P.face_cycle_succ(e3));
P.init_maps(e4,P.get_a(e3)-90,"1",false,false);
P.init_maps(REV(e4),90,"0",true,false);
edge e5= P.split_bend edge(e4);
P.set_a(e1,P.get_a(e1)-90);
P.set_inner(e2,true);
P.init_maps(e3,90,EPS,true,false);
P.set_type(source(e5),dissection);
S.push(e1);
size-= 2;state= 0;

}
e= P.succ_corner_edge(e);
}

}
//if(check&&!P.check())error_handler(1,ERR_NO_ORTHO_REP);

P.init_rest();
P.assign_directions(P.first_edge(),north);

```

```

#ifdef CPLEX

```

```

int n= 0;int m= 0;
node_array<int>v_num(P);edge_array<int>e_num(P,-1);
forall_nodes(v,P)v_num[v]= n++;
forall_edges(e,P)if(P.get_dir(e)==north||P.get_dir(e)==west)
e_num[e]= m++;
int basic_rows= 2*m;
int rows= basic_rows;
int cols= 2*n;
int basic_nonzeroes= 2*basic_rows;
int nonzeroes= basic_nonzeroes;
double*obj= new double[cols];
double*rhs= new double[rows];
char*sense= new char[rows];
int*matbeg= new int[cols];
int*matcnt= new int[cols];
int*matind= new int[nonzeroes];
double*matval= new double[nonzeroes];
double*lb= new double[cols];
double*ub= new double[cols];

```

```

CPXENVptr env= NULL;
CPXLPptr lp= NULL;

```

```

int status;

```

```

for(int j= 0;j<cols;j++){
obj[j]= 0.0;
matbeg[j]= 0;matcnt[j]= 0;
lb[j]= 0.0;ub[j]= INFBOUND;
}
for(int k= 0;k<basic_nonzeroes;k++){
matind[k]= 0;matval[k]= 0.0;
}

```

```

#define X(v) (v_num[v])
#define Y(v) (n + v_num[v])
forall_edges(e,P){
node s= source(e),t= target(e);
switch(P.get_dir(e)){
case north:
obj[v_num[s]]-= 1.0;
obj[v_num[t]]+= 1.0;
break;
case west:
obj[v_num[s]+n]-= 1.0;
obj[v_num[t]+n]+= 1.0;
break;
default:break;
}
}

```

```

int nonzero_cnt= 0;
forall_nodes(v,P){
int act_col= X(v);
matbeg[act_col]= nonzero_cnt;
forall_inout_edges(e,v){
bool out_edge= (v==source(e));int act_row;
switch(P.get_dir(e)){
case north:
act_row= 2*e_num[e];
matcnt[act_col]++;
if(out_edge)matval[nonzero_cnt]= 1.0;
else matval[nonzero_cnt]= -1.0;
matind[nonzero_cnt]= act_row;
sense[act_row]= 'L';rhs[act_row]= -1.0;
++nonzero_cnt;
break;
case west:
act_row= 2*e_num[e]+1;
matcnt[act_col]++;
if(out_edge)matval[nonzero_cnt]= 1.0;
else matval[nonzero_cnt]= -1.0;
matind[nonzero_cnt]= act_row;
sense[act_row]= 'E';rhs[act_row]= 0.0;
++nonzero_cnt;
break;
default:break;
}
}
}
forall_nodes(v,P){
int act_col= Y(v);
matbeg[act_col]= nonzero_cnt;
forall_inout_edges(e,v){
bool out_edge= (v==source(e));int act_row;
switch(P.get_dir(e)){
case west:
act_row= 2*e_num[e];
matcnt[act_col]++;
if(out_edge)matval[nonzero_cnt]= 1.0;
else matval[nonzero_cnt]= -1.0;
matind[nonzero_cnt]= act_row;
sense[act_row]= 'L';rhs[act_row]= -1.0;
++nonzero_cnt;
break;
case north:
act_row= 2*e_num[e]+1;
matcnt[act_col]++;
if(out_edge)matval[nonzero_cnt]= 1.0;
else matval[nonzero_cnt]= -1.0;
matind[nonzero_cnt]= act_row;
sense[act_row]= 'E';rhs[act_row]= 0.0;
++nonzero_cnt;
break;
default:break;
}
}
}

```

```

}
env= CPXopenCPLEX(&status);
if (env==NULL)error_handler(1,ERR_OPEN_CPLEX_ENV);

lp= CPXloadlp(env,"find_coords",cols,basic_rows,CPX_MIN,obj,rhs,
sense,matbeg,matcnt,matind,
matval,lb,ub,NULL,cols,rows,nonzeroes);
if (lp==NULL)error_handler(1,ERR_LOAD_LP);


int solstat;
double objval;
double*sol= new double[cols];
double*pi= new double[rows];
double*slack= new double[rows];
double*dj= new double[cols];

status= CPXoptimize(env,lp);
if(status){
CPXcloseCPLEX(&env);
error_handler(1,ERR_CPLEX_OPT);
}

status= CPXsolution(env,lp,&solstat,&objval,sol,pi,slack,dj);
if(status)error_handler(1,ERR_CPLEX_GET_SOLUTION);
int max_x= 0,max_y= 0;
forall_nodes(v,P)if(P.get_type(v)==real){
max_x= Max(max_x,(x_pos[P.get_orig(v)]=(int)sol[X(v)]));
max_y= Max(max_y,(y_pos[P.get_orig(v)]=(int)sol[Y(v)]));
}

if(env)status= CPXfreeprob(env,&lp);
if(status)error_handler(1,ERR_CPLEX_FREE);
status= CPXcloseCPLEX(&env);
if(status)error_handler(1,ERR_CPLEX_CLOSE);


#else


graph N_h,N_v;
face_map<node>FtoN_h(P),FtoN_v(P);
edge_map<edge>EtoA(P);


node s_h= N_h.new_node(),t_h= N_h.new_node(),s_v= N_v.new_node(),
t_v= N_v.new_node();

forall_faces(f,P){
if(!IS_OUTER(f)){
FtoN_h[f]= N_h.new_node();

```

```

FtoN_v[f]= N_v.new_node();
}
}

```

```

forall_faces(f,P)forall_face_edges(e,f){
face g= P.face_of(REV(e));
bool out_f= IS_OUTER(f),out_g= IS_OUTER(g);
switch(P.get_dir(e)){
case north:
if(out_f&&!out_g)EtoA[e]= N_h.new_edge(FtoN_h[g],t_h);
if(out_g&&!out_f)EtoA[e]= N_h.new_edge(s_h,FtoN_h[f]);
if(out_f&&out_g)EtoA[e]= N_h.new_edge(s_h,t_h);
if(!out_f&&!out_g)EtoA[e]=
N_h.new_edge(FtoN_h[g],FtoN_h[f]);
break;
case west:
if(out_f&&!out_g)EtoA[e]= N_v.new_edge(FtoN_v[g],t_v);
if(out_g&&!out_f)EtoA[e]= N_v.new_edge(s_v,FtoN_v[f]);
if(out_f&&out_g)EtoA[e]= N_v.new_edge(s_v,t_v);
if(!out_f&&!out_g)EtoA[e]=
N_v.new_edge(FtoN_v[g],FtoN_v[f]);
break;
default:break;
}
}
edge e_h= N_h.new_edge(s_h,t_h);
edge e_v= N_v.new_edge(s_v,t_v);

```

```

edge_array<int>cap_h(N_h,INFINITY),cap_v(N_v,INFINITY);
edge_array<int>cost_h(N_h,1),cost_v(N_v,1);
edge_array<int>l_h(N_h,1),l_v(N_v,1);
edge_array<int>flow_h(N_h,0),flow_v(N_v,0);
node_array<int>b_h(N_h,0),b_v(N_v,0);
forall_nodes(v,N_h)
b_h[s_h]+= abs(outdeg(v)-indeg(v));
b_h[s_h]= b_h[s_h]/2+1;
b_h[t_h]= -b_h[s_h];
forall_nodes(v,N_v)
b_v[s_v]+= abs(outdeg(v)-indeg(v));
b_v[s_v]= b_v[s_v]/2+1;
b_v[t_v]= -b_v[s_v];
cost_h[e_h]= 0;cost_v[e_v]= 0;
l_h[e_h]= 0;l_v[e_v]= 0;

```

```

feasible= MIN_COST_FLOW(N_h,l_h,cap_h,cost_h,b_h,flow_h);
if(!feasible)error_handler(1,ERR_NO_FEASIBLE_FLOW);
feasible= MIN_COST_FLOW(N_v,l_v,cap_v,cost_v,b_v,flow_v);
if(!feasible)error_handler(1,ERR_NO_FEASIBLE_FLOW);

```

```

forall_edges(e,P){
switch(P.get_dir(e)){
case north:
P.set_length(e,flow_h[EtoA[e]]);
P.set_length(REV(e),flow_h[EtoA[e]]);
break;
case west:
P.set_length(e,flow_v[EtoA[e]]);
P.set_length(REV(e),flow_v[EtoA[e]]);
break;
default:break;
}
}

//for(v= P.choose_node();P.get_type(v)!=real;v= P.succ_node(v));

forall_nodes(v,P)
    if (P.get_type(v) == real) break;

x_pos.init(G);y_pos.init(G);node_array<bool>seen(P,false);
P.determine_position(v,0,0,seen);
P.norm_positions();
int max_x= 0,max_y= 0;
forall_nodes(v,P)if(P.get_type(v)==real){
int x = P.get_x(v);
int y = P.get_y(v);
x_pos[P.get_orig(v)]=x;
y_pos[P.get_orig(v)]=y;
max_x= Max(max_x,x);
max_y= Max(max_y,y);
}

//cout<<"coordinates computed by network method"<<endl;

#endif

forall_nodes(v,P)if(P.get_type(v)==big){
int x_big,y_big;
list<int>lx,ly;
face f_cage= P.face_of(corr_cage_edge[v]);
forall_face_edges(e,f_cage){
if(P.get_type(source(e))!=dissection){
int x= P.get_x(source(e)),y= P.get_y(source(e));
lx.append(x);ly.append(y);
}
}
lx.sort();ly.sort();list_item it;
x_big= (lx.head()+lx.tail())/2;

```



```

y_big= (ly.head()+ly.tail())/2;

forall_items(it,lx)
    if(lx[it]!=lx.head()&&lx[it]!=lx.tail()
        &&lx[it]==lx[lx.cyclic_succ(it)]) x_big= lx[it];

forall_items(it,ly)
    if(ly[it]!=ly.head()&&ly[it]!=ly.tail()
        &&ly[it]==ly[ly.cyclic_succ(it)]) y_big= ly[it];

x_pos[P.get_orig(v)]= x_big;
y_pos[P.get_orig(v)]= y_big;
}

x_bends.init(G);y_bends.init(G);
forall_edges(e,G)
    forall(v,b_nodes[e]){
x_bends[e].append(P.get_x(v));
y_bends[e].append(P.get_y(v));
}

//cout<<"cages resolved"<<endl<<endl;

return Max(max_x,max_y);
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _spring.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
+ *****/
#include <LEDA/graph_alg.h>

#include <math.h>
#include <LEDA/array2.h>

#define FREPULSE(d) ((k2 > d) ? kk/d : 0)

static float log_2(int x)
{ float l = 0;
  while (x)
  { l++;
    x >>= 1;
  }
  return l/2;
}

void SPRING_EMBEDDING(const graph& G, node_array<double>& xpos,
                     node_array<double>& ypos,
                     double xleft, double xright,
                     double ybottom, double ytop,
                     int iterations)
{ list<node> L;
  SPRING_EMBEDDING(G,L,xpos,ypos,xleft,xright,ybottom,ytop,iterations);
}

void SPRING_EMBEDDING(const graph& G, const list<node>& fixed_nodes,
                     node_array<double>& xpos,
                     node_array<double>& ypos,
                     double xleft, double xright,
                     double ybottom, double ytop,
                     int iterations)
{
  if (xleft >= xright || ybottom >= ytop)
    error_handler(1,"SPRING_EMBDDING: illegal bounds.");

  node_array<list_item> lit(G);
  node_array<bool> fixed(G,false);
}

```

```

node u,v;
edge e;

forall(v,fixed_nodes) fixed[v] = true;

int c_f = 1;

double width  = xright - xleft;
double height = ytop - ybottom;

double tx_null = width/50;
double ty_null = height/50;
double tx = tx_null;
double ty = ty_null;

double k = sqrt(width*height / G.number_of_nodes()) / 2;
double k2 = 2*k;
double kk = k*k;

int ki = int(k);

if (ki == 0) ki = 1;

//build matrix of node lists

int xA = int(width / ki + 1);
int yA = int(height / ki + 1);

array2<list<node> > A(-1,xA,-1,yA);

forall_nodes(v,G)
{ int i = int((xpos[v] - xleft) / ki);
  int j = int((ypos[v] - ybottom) / ki);
  if (i >= xA || i < 0) error_handler(1,"spring: node out of range");
  if (j >= yA || j < 0) error_handler(1,"spring: node out of range");
  lit[v] = A(i,j).push(v);
}

while (c_f < iterations)
{
  node_array<double> xdisp(G,0);
  node_array<double> ydisp(G,0);

  // repulsive forces

  forall_nodes(v,G)
  { int i = int((xpos[v] - xleft) / ki);
    int j = int((ypos[v] - ybottom) / ki);

    double xv = xpos[v];
    double yv = ypos[v];

    for(int m = -1; m <= 1; m++)
      for(int n = -1; n <= 1; n++)
        forall(u,A(i+m,j+n))
        { if(u == v) continue;
          double xdists = xv - xpos[u];
          double ydists = yv - ypos[u];
          double dist = sqrt(xdists * xdists + ydists * ydists);
          if (dist < 1e-3) dist = 1e-3;
        }
      }
    }
  }
  c_f++;
}

```

```

        xdisp[v] += FREPULSE(dist) * xdist / dist;
        ydisp[v] += FREPULSE(dist) * ydist / dist;
    }
    xdisp[v] *= (double(rand_int(750,1250))/1000.0);
    ydisp[v] *= (double(rand_int(750,1250))/1000.0);
}

// attractive forces
forall_edges(e,G)
{ node u = G.source(e);
  node v = G.target(e);
  double xdist=xpos[v]-xpos[u];
  double ydist=ypos[v]-ypos[u];
  double dist=sqrt(xdist*xdist+ydist*ydist);

  float f = (G.degree(u)+G.degree(v))/6.0;

  dist /= f;

  xdisp[v]-=xdist*dist/k;
  ydisp[v]-=ydist*dist/k;
  xdisp[u]+=xdist*dist/k;
  ydisp[u]+=ydist*dist/k;
}

// preventions
forall_nodes(v,G)
{
    if (fixed[v]) continue;

    int i0 = int((xpos[v] - xleft)/ki);
    int j0 = int((ypos[v] - ybottom)/ki);

    double xd= xdisp[v];
    double yd= ydisp[v];
    double dist = sqrt(xd*xd+yd*yd);

    if (dist < 1) dist = 1;

    xd = tx*xd/dist;
    yd = ty*yd/dist;

    double xp = xpos[v] + xd;
    double yp = ypos[v] + yd;

    int i = i0;
    int j = j0;

    if (xp > xleft && xp < xright)
    { xpos[v] = xp;
      i = int((xp - xleft)/ki);
    }

    if (yp > ybottom && yp < ytop)
    { ypos[v] = yp;
      j = int((yp - ybottom)/ki);
    }
}

```

```

    }

    if (i != i0 || j != j0)
    { if (lit[v] == nil) error_handler(1,"delete nil item");
      A(i0,j0).del_item(lit[v]);
      lit[v] = A(i,j).push(v);
    }

    }

    tx = tx_null / log_2(c_f);
    ty = ty_null / log_2(c_f);

    c_f++;
  }
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _sugiyama.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
+ *****/
//-----
--
// SUGIYAMA EMBEDDING
//
// D. Ambras (1996/97)
//-----
--

#include <stdio.h>
#include <LEDA/array.h>
#include <LEDA/p_queue.h>
#include <LEDA/stream.h>
#include <LEDA/graph.h>
#include <LEDA/graph_alg.h>

#ifdef _DRAW_SUGI
#include <LEDA/panel.h>
#endif

typedef int bitarray;

#define SORT_LOWER 1
#define SORT_UPPER 2
#define SORT_BOTH 3
#define PERMUTE 4
#define IMPROVE 8

#define HIGH 1
#define LOW 0
#define UP 1
#define DOWN 0

#ifdef _DRAW_SUGI
window W;
#endif
double aspect_ratio=3;
int ps_count=1;
int perm_max=10, sort_max=6;
int max_level;
bool do_debug=true;

```

```

void write_to_ps(const graph &G, array<list<node> > &Level,
node_array<int>
                &x_coord, node_array<int> &nlabel, int width, int
height,
                char *headline, file_ostream &out)
{
    int    x_off, y_off, x_gap, y_gap, radius, i;
    x_off  = y_off= x_gap= 20;
    y_gap  = int(x_gap * aspect_ratio + 0.5);
    radius = 3;

    node    v, s, t;
    edge    e;

    out << "!PS\n";
    out << "%Creator: SUGIYAMA_EMBEDDING v1.0\n";
    out << "%Pages: 1\n";
    out << "%PageOrder: Ascend\n";
    out << "%BoundingBox: " << 0 << " " << 0 << " ";
    out << x_gap*width + 2*x_off << " " << y_gap*height + 2*y_off << "\n";
    out << "%EndComments\n\n%Page: 1 1\n\n";

    out << "% " << headline << endl << endl;

    out << "% drawing the nodes\n\n";
    i=-1;
    while ((++i) <= max_level)
        forall(v, Level[i]) if (nlabel[v] != 0)
        { out << "newpath ";
          out << x_gap * x_coord[v] + x_off << " " << y_gap * i + y_off;
          out << " " << radius << " 0 360 arc fill\n";
        }

    out << "\n% drawing the edges\n\nnewpath\n";
    i=0;

    while ((++i) <= max_level) // drawing the
edges
        forall(v, Level[i-1])
            forall_out_edges(e, v)
            { s= G.source(e); t= G.target(e);
              out << x_gap * x_coord[s] + x_off << " " << y_gap*(i-1) + y_off
                << " moveto ";
              out << x_gap * x_coord[t] + x_off << " " << y_gap*i + y_off
                << " lineto\n";
            }

    out << "stroke\nshowpage\n\n%%EOF" << endl << flush;
}

```

```

#ifdef _DRAW_SUGI

void user_exit(void)
{
    cout << "Aborted." << endl << flush;
    exit(1);
}

bool draw_hierarchy(const graph &G, array<list<node> > &Level,
                    node_array<int> &x_coord, node_array<int> &nlabel,
                    char *headline, const bool &draw_virtual_nodes)
{
    int    i, x_min=100, x_max=-100, xv, b=NO_BUTTON;
    int    width=0, height;
    double W_size;
    color  bkg_color=white, node_color, edge_color=blue;
    color  rnode_color=green, vnode_color=white;
    node   v, w;
    edge   e;
    char   filename[200];

    forall_nodes(v, G)
    { i=x_coord[v];
      if (x_min > i) x_min= i;
      if (x_max < i) x_max= i;
    }

    width= x_max -x_min +1;
    height=max_level + 1;

    W_size=(width > height*aspect_ratio ? width: height*aspect_ratio);
    W.init(-1, W_size, -1);
    W.clear(bkg_color);  W.set_frame_label(headline);

#ifdef _DEBUG_SUGI
    if (do_debug)
    { W.message("writing graph to disk ...");
      sprintf(filename, "figure%d.ps", ps_count);
      file_ostream ps_out(filename);
      write_to_ps(G, Level, x_coord, nlabel, width,
                  height, filename, ps_out);
      ps_count++;
    }
#endif
#ifdef _DEBUG2_SUGI
    if (do_debug)
    { W.message("writing graph to disk ...");
      sprintf(filename, "figure%d.ps", ps_count);
      file_ostream ps_out(filename);
      write_to_ps(G, Level, x_coord, nlabel, width,
                  height, filename, ps_out);
      ps_count++;
    }
}

```



```

#endif

    i=0; forall(v, Level[i])
        W.draw_int_node(x_coord[v]-x_min, i*aspect_ratio, nlabel[v],
            rnode_color);

    do
    { if (i == max_level)                                // no more levels
      { W.message(
        "press left mouse button to proceed, other button to abort");
        while (b == NO_BUTTON) b= W.get_button();
        W.del_messages();
        if (b==MOUSE_BUTTON(1))
        { W.message("continue calculations, please wait ...");
          return true;
        }
        return false;
      }

      forall(v, Level[i])                                // drawing the
edges
      { xv=x_coord[v]-x_min;
        forall_out_edges(e, v)
        if (draw_virtual_nodes || (nlabel[v] > 0 && nlabel[G.target(e)] >
0) )
            W.draw_edge(xv, i*aspect_ratio, x_coord[G.target(e)]-x_min,
                (i+1)*aspect_ratio, edge_color);
            else
            W.draw_segment(xv, i*aspect_ratio, x_coord[G.target(e)]-x_min,
                (i+1)*aspect_ratio, edge_color);
        }

      forall(v, Level[i+1])                                // drawing the
nodes
      { if (nlabel[v]==0) node_color=vnode_color;
        else node_color=rnode_color;
        if (draw_virtual_nodes || nlabel[v] > 0)
            W.draw_int_node(x_coord[v]-x_min, (i+1)*aspect_ratio,
                nlabel[v], node_color);
        }
        i++;
    } while (0<1);
}

#endif

bool slide_node(const graph &G, const node &v, const int &priority,
    const int &best_pos, node_array<int> &x_coord, node_array<int>
&x_prio,
    array<bool> &x_set, array<node> &x_owner)
{
    int    cur_pos=x_coord[v], N=G.number_of_nodes();

```

```

while (cur_pos < best_pos)
{ if (cur_pos > N-1) return false;

  if (x_set[cur_pos+1])
  { if (x_prio[ x_owner[cur_pos+1]] > priority)
    return false;

    if ( !slide_node(G, x_owner[cur_pos+1],
                      priority, cur_pos+2,
                      x_coord, x_prio,
                      x_set, x_owner)) return false;
                                     // owner was move-able, but
others not
  }

  x_set[cur_pos++]=false;
  x_set[cur_pos]=true;           // move node one step
  x_owner[cur_pos]=v;
  x_coord[v]=cur_pos;
}

while (cur_pos > best_pos)
{ if (cur_pos < -N+1) return false;

  if (x_set[cur_pos-1])
  { if (x_prio[ x_owner[cur_pos-1]] >= priority)
    return false;

    if ( !slide_node(G, x_owner[cur_pos-1],
                      priority, cur_pos-2,
                      x_coord, x_prio,
                      x_set, x_owner)) return false;
  }
  x_set[cur_pos--]=false;
  x_set[cur_pos]=true;
  x_owner[cur_pos]=v;
  x_coord[v]=cur_pos;
}

return true;                     // successful move
}

bool two_level(const graph &G, array<list<node> > &Level, const int
&upper,
               node_array<int> &x_coord, node_array<int> &x_prio,
               const bitarray &what_to_do)
{
  bool      changes=false;
  int       lower=upper+1, N=G.number_of_nodes();

```

```

int          x, count, pcount, hprio_count;
node         nu, nl;
node         u, v;
edge         e;
double       bary, old_bary, hprio_bary;

list<node>    equal_bary;
p_queue<double, node> Su, Sl;
pq_item      pit;

array<bool>   x_set(-N, N);
array<node>   x_owner(-N, N);
node_array<int> p_bary(G);

if (what_to_do & SORT_UPPER)
{ forall(nu, Level[upper])
  { bary=0;
    hprio_count=0;
    hprio_bary=0;

    forall_out_edges(e, nu)
    { x=x_coord[ G.target(e) ]; bary+=x;
      if (x_prio[ G.target(e) ] == N) // a long edge
      { hprio_count++;
        hprio_bary+=x;
      }
    }

    if (count=G.outdeg(nu)) bary =bary/count;
    else bary=x_coord[nu]; // no in- edges

    if (what_to_do & IMPROVE)
    { Su.insert(-x_prio[nu], nu); // sort by priority
      p_bary[nu]=(int) (bary+0.5);
      if (hprio_count)
        p_bary[nu]=(int) (hprio_bary/ hprio_count +0.5);
      // we ignore short edges
    }
    else Su.insert(bary, nu);
  }

  if (what_to_do & IMPROVE)
  { for(x=-N; x<= N; x++) x_set[x]=false;
    forall(v, Level[upper])
    { x_owner[ x_coord[v]]=v;
      x_set[ x_coord[v]]=true;
    }
  }

  count=0; old_bary=-1; pcount=-2;

  while(Su.size())
  { pit=Su.find_min();
    v=Su.inf(pit);
    bary=Su.prio(pit);
    Su.del_item(pit);
  }
}

```

```

        if (what_to_do & IMPROVE)                                // set real
x_coordinates
        slide_node(G, v,
                    x_prio[v], p_bary[v],
                    x_coord, x_prio,
                    x_set, x_owner);

        else if (x_coord[v] != count)                            // set consecutive
x_coordinates
        { changes=true;
          x_coord[v] = count;
        }

        if (what_to_do & PERMUTE)                                // we shall permute
        { if (bary==old_bary)
          { equal_bary.append(u);
            pcount=count;
          }
          else
          { if (pcount == count-1) equal_bary.append(u);
            while (!equal_bary.empty())
            { changes=true;
              x_coord[equal_bary.pop()]= pcount--;
            }
          }
          count++;
          old_bary=bary;
          u=v;
        }
        if (pcount == count-1) equal_bary.append(u);

        while (!equal_bary.empty())                              // flush equal_bary
        { changes=true;
          x_coord[equal_bary.pop()]= pcount--;
        }
    }

    if (what_to_do & SORT_LOWER)
    { forall(nl, Level[lower])
      { bary=0;
        hprio_count=0;
        hprio_bary=0;

        forall_in_edges(e, nl)
        { x=x_coord[ G.source(e) ]; bary+=x;
          if (x_prio[ G.source(e) ] == N)                // a long edge
          { hprio_count++;
            hprio_bary+=x;
          }
        }
        if (count=G.indeg(nl)) bary =bary/count;
        else bary=x_coord[nl];

        if (what_to_do & IMPROVE)                            // set real
x_coordinates
        { Sl.insert(-x_prio[nl], nl);
          p_bary[nl]=(int)(bary +0.5);
          if (hprio_count)

```

```

        p_bary[nl]=(int) (hprio_bary/ hprio_count +0.5);
                                                // ignore short edges
    }
    else Sl.insert(bary, nl);
}

if (what_to_do & IMPROVE)
{ for(x=-N; x<= N; x++) x_set[x]=false;
  forall(v, Level[lower])
  { x_owner[ x_coord[v]]= v;
    x_set[ x_coord[v]]= true;
  }
}

count=0; old_bary=-1; pcount=-2;

while(Sl.size())
{ pit=Sl.find_min();
  v=Sl.inf(pit);
  bary=Sl.prio(pit);
  Sl.del_item(pit);

  if (what_to_do & IMPROVE)
    slide_node(G, v,
               x_prio[v], p_bary[v],
               x_coord, x_prio,
               x_set, x_owner);

  else if (x_coord[v] != count)
  { changes=true;
    x_coord[v] = count;
  }
  if (what_to_do & PERMUTE)
  if (bary==old_bary)
  { equal_bary.append(u);
    pcount=count;
  }
  else
  { if (pcount == count-1) equal_bary.append(u);
    while (!equal_bary.empty())
    { changes=true;
      x_coord[equal_bary.pop()]=pcount--;
    }
  }
  count++;
  old_bary=bary;
  u=v;
}
if (pcount == count-1) equal_bary.append(u);

while (!equal_bary.empty()) // flush equal_bary
{ changes=true;
  x_coord[equal_bary.pop()]=pcount--;
}
}

#ifdef _DEBUG2_SUGI

```

```

char    headline[200], dir[10], text_p[10], text_i[10];

if (( upper > 10) && do_debug)
{ if (what_to_do & SORT_LOWER) sprintf(dir, "[down]");
  else sprintf(dir, "[up]");
// ( supposed SORT_BOTH is not selected )

  if (what_to_do & PERMUTE) sprintf(text_p, " + permute");
  else sprintf(text_p, " ");

  if (what_to_do & IMPROVE) sprintf(text_i, " + improve");
  else sprintf(text_i, " ");

  sprintf(headline, "after two_level for u-level %d, sort %swards%s%s",
    upper, dir, text_p, text_i);

  if (!draw_hierarchy(G, Level, x_coord, x_coord, headline, true))
    user_exit();
}
#endif

return changes;
}

list<edge> sort_edges(const graph &G, array<list<node> > &Level,
                     const int &column, const int &height,
                     node_array<int> &x_coord, list<edge> &edges)
{
  int      i=0;
  node     v;
  edge     e;
  list<edge> result;

  array< list<edge> >  nlist( Level[column].size() );

  forall(e, edges)
  { if (height == LOW)  v=G.target(e);
    else v=G.source(e);
    nlist[ x_coord[v] ].append(e);
  }

  while (i < Level[column].size() )
    result.conc(nlist[i++]);

  return result;
}

static bool cross(const graph &G, node_array<int> &x_coord,
                  edge_array<int> &current_pos, const edge &e, const
edge &f)
{
  int    xu1, xu2, xl1, xl2;

  xu1=current_pos[e];
  xu2=current_pos[f];
  xl1=x_coord[ G.target(e) ];

```

```

xl2=x_coord[ G.target(f) ];

return ( xu1 < xu2 && xl1 > xl2) || ( xu1 > xu2 && xl1 < xl2 );
}

int number_of_crossings(const graph &G, array<list<node> > &Level,
                        const int &upper, node_array<int> &x_coord)
{
    int          lower=upper+1, result=0, i;
    node         vu;
    edge         e, f, g, h;
    list<edge>    edges, work_list;
    list_item     lit;

    forall(vu, Level[upper])
        forall_adj_edges(e, vu) edges.append(e);

    edges= sort_edges(G, Level, lower, LOW , x_coord, edges);
    edges= sort_edges(G, Level, upper, HIGH, x_coord, edges);

    array<edge>          current_order(edges.size() );
    edge_array<int>      current_pos(G);
    edge_array<list_item> worklist_it(G, nil);

    i=0;
    forall(e, edges)
    { current_order[i]=e;
      current_pos[e]=i++;
    }

    for (i=0; i < edges.size()-1; i++)
    { e=current_order[i];
      f=current_order[i+1];
      if ( cross(G, x_coord, current_pos, e, f) )
          worklist_it[e] = work_list.append(e);
    }

    while (!work_list.empty() )
    { e=work_list.pop();
      worklist_it[e]= nil;
      i=current_pos[e];
      result++;

      if (i < edges.size()-1)
      { f=current_order[i+1];
        current_order[i]=f;
        current_order[i+1]=e;
        current_pos[e]++;
        current_pos[f]--;

        if (lit=worklist_it[f])

```

```

    { work_list.del_item(lit);
      worklist_it[f]= nil;
    }

// Test, whether e or f cross their new neighbor:

    if (i>0)
    { g=current_order[i-1];
      if (cross(G, x_coord, current_pos, g, f) )
      { if (!worklist_it[g])
          worklist_it[g]= work_list.append(g);
        }
      else
      if (lit=worklist_it[g])
      { work_list.del_item(lit);
        worklist_it[g]= nil;
      }
    }

    if (i < edges.size()-2)
    { h=current_order[i+2];
      if (cross(G, x_coord, current_pos, e, h) )
        worklist_it[e]=work_list.append(e);
    }
  }
}

return result;
}

void init_positions(graph &G, array<list<node> > &Level,
                   node_array<int> &x_coord, node_array<int> &nlabel,
                   node_array<int> &x_prio, bool first)
{
  node    v;
  int     i=-1, label=1, x, N=G.number_of_nodes();

  while ( (++i) <= max_level)
  { x=0;
    forall(v, Level[i])
    { if (first) x_coord[v]=x++;
      if (x_prio[v]== -1)
      { x_prio[v]= N;                               // dummy node = highest priority
        nlabel[v]= 0;
      }
      else
      { x_prio[v]= G.degree(v);
        nlabel[v]= label++;
      }
    }
  }
}

bool make_hierarchy(graph &G, node_array<int> &the_level,
array<list<node> >

```



```

                                &Level, list<node> &dummy_nodes, bool first)
{
    int    i=0;
    node    v;

    forall_nodes(v, G)
        Level[ the_level[v] ].append(v);

    if (!first) return false;


    bool    lost_edge=false;
    int    j;
    node    w, a, b;
    edge    e;
    list<edge> remove, split, turn;

    Make_Simple(G);

    forall_nodes(v, G)
    { i=the_level[v];
      forall_out_edges(e, v)
      { j=the_level[ G.target(e) ];
        if (i>j)    turn.append(e);
        if (i==j)  remove.append(e);
        if (i+1<j) split.append(e);
      }
    }

    forall(e, turn)
    {
        a=G.source(e);    b=G.target(e);

        e=G.rev_edge(e);

        if (the_level[b]+1 < the_level[a]) split.append(e);
    }

    if (!remove.empty())
    { error_handler(1,"input is not a hierarchical graph.");
      lost_edge=true;
      forall(e, remove)  G.del_edge(e);
    }


    dummy_nodes.clear();

    forall(e,split)
    {
        a = G.source(e);
        b = G.target(e);

        i = the_level[a] + 1;
        j = the_level[b];

        w = G.new_node();
        Level[i].append(w);
    }

```

```

    dummy_nodes.append(w);
    G.move_edge(e,a,w);
    a = w;
    i++;

    while (i<j)
    { w=G.new_node();
      Level[i].append(w);
      dummy_nodes.append(w);
      G.new_edge(a,w);
      a = w;
      i++;
    }

    G.new_edge(a,b);
}

if (split.empty() ) return lost_edge;

the_level.init(G); // there are more nodes now
i=-1;
while( (++i) <= max_level)
    forall(v, Level[i]) the_level[v]=i;

return lost_edge;
}

inline int all_crossings(const graph &G, array<list<node> > &Level,
                        node_array<int> &x_coord, array<int> &L_crosses)
{
    int i=0, crosses=0, c;

    while(i < max_level)
    { c= L_crosses[i]= number_of_crossings(G, Level, i++, x_coord);
      crosses+=c;
    }

    return crosses;
}

inline void copy_all_xcoord(const graph &G, array<list<node> >& /* Level
*/,
                        node_array<int> &x_coord, node_array<int>
&x_new,
                        array<int> &L_crosses, array<int>
&L_crosses_new)
{
    int i=0;
    node v;

    while(i < max_level)
    { L_crosses_new[i]= L_crosses[i];

```

```

    i++;
}
forall_nodes(v, G) x_new[v]=x_coord[v];
}

bool update_best(const graph &G, array<list<node> > &Level,
                node_array<int> &x_coord, node_array<int> &x_new,
                array<int> &L_crosses, const int &i,
                int &crosses, const bitarray &what_to_do)

{ bool update, process_next;
  int rcn1, rcn2=0, next_level, cr_sum, work_level;
  // int make_worse= 1.2;
  node v;

  two_level(G, Level, i, x_coord, x_coord, what_to_do);
                                     // makes the requested change

  if (what_to_do & SORT_LOWER)
  { next_level=i+1;
    work_level=i+1;
  }
  else
  { next_level=i-1;
    work_level=i;
  }

  process_next=((what_to_do & SORT_LOWER) && (next_level < max_level));
  if ((what_to_do & SORT_UPPER) && (next_level > -1))
process_next=true;
                                     // what level is neighbor to the changed
if any

  rcn1= number_of_crossings(G, Level, i, x_coord);
  if (process_next)
    rcn2= number_of_crossings(G, Level, next_level, x_coord);

  cr_sum= L_crosses[i];
  if (process_next) cr_sum+= L_crosses[next_level];

  update= ((rcn1+rcn2) <= cr_sum);
  // update= ((rcn1+rcn2) <= cr_sum* make_worse);

  if (update)
  { crosses= crosses- L_crosses[i]+ rcn1;
    L_crosses[i]= rcn1;
    forall(v, Level[work_level]) x_new[v]=x_coord[v];

    if (process_next)
    { crosses= crosses- L_crosses[next_level]+ rcn2;
      L_crosses[next_level]= rcn2;
    }
  }
  else
    forall(v, Level[work_level]) x_coord[v]=x_new[v];
}

```

```

return update;
}

```

```

bool down_up_sort(graph &G, array<list<node> > &Level, node_array<int>&
x_coord,

```

```

    array<int> &L_crosses, int &crosses,
    node_array<int>& /* nlabel */,
    int how_often, int first, int begin)

```

```

{
    int    i=0, j=0;
    int    best_crosses, orig_crosses;
    char    headline[200];
    bool    ch, u_changes=false, l_changes=false;
    bool    result=false;

    node_array<int>    x_new(G);
    node_array<int>    x_old(G);
    array<int>    L_crosses_new(max_level);
    array<int>    L_crosses_old(max_level);
    copy_all_xcoord(G, Level, x_coord, x_old, L_crosses, L_crosses_old);

    array<bool> Lu_changes(max_level);
    array<bool> Ll_changes(max_level);
    for(i=0; i < max_level; i++)
        Lu_changes[i]=Ll_changes[i]=false;

    best_crosses= orig_crosses= crosses;

```

```

    j=how_often;
    while (j-->0)
    { if (best_crosses==0) break;
      if (first==DOWN)
      { for (i=begin; i< max_level; i++)
        { ch=two_level(G, Level, i, x_coord, x_coord, SORT_LOWER);
          if (ch && !Ll_changes[i]) l_changes=true;
          Ll_changes[i]=ch;
          if (ch)
          { crosses-= L_crosses[i];
            L_crosses[i]= number_of_crossings(G, Level, i, x_coord);
            crosses+= L_crosses[i];
            if (i< max_level-1)
            { crosses-= L_crosses[i+1];
              L_crosses[i+1]= number_of_crossings(G, Level, i+1, x_coord);
              crosses+= L_crosses[i+1];
            }
            if (crosses < best_crosses)
            { copy_all_xcoord(G, Level, x_coord, x_new,
                          L_crosses, L_crosses_new);
              best_crosses=crosses;                // new high score
            }
          }
        }
      }
    }
}

```

```

    }
    begin=max_level-1; // set 'begin' for up-
sort new
    }
    if (best_crosses==0) break;

for (i=begin; i>=0; i--)
{ ch=two_level(G, Level, i, x_coord, x_coord, SORT_UPPER);
  if (ch && !Lu_changes[i]) u_changes=true;
  Lu_changes[i]=ch;
  if (ch)
  { crosses-= L_crosses[i];
    L_crosses[i]= number_of_crossings(G, Level, i, x_coord);
    crosses+= L_crosses[i];
    if (i>0)
    { crosses-= L_crosses[i-1];
      L_crosses[i-1]= number_of_crossings(G, Level, i-1, x_coord);
      crosses+= L_crosses[i-1];
    }
    if (crosses < best_crosses)
    { copy_all_xcoord(G, Level, x_coord, x_new, L_crosses,
L_crosses_new);
      best_crosses=crosses;
    }
  }
}

first=DOWN;
begin=0;

if ( !(u_changes || l_changes) ) break; else result=true;

u_changes= l_changes= false;
}

if (best_crosses < orig_crosses)
{ copy_all_xcoord(G, Level, x_new, x_coord, L_crosses_new, L_crosses);
  crosses=best_crosses;
}
if (crosses > orig_crosses)
{ copy_all_xcoord(G, Level, x_old, x_coord, L_crosses_old, L_crosses);
  crosses=orig_crosses;
}
return result;
}

```

```

int down_up_change(graph &G, array<list<node> > &Level, node_array<int>
&x_coord, node_array<int> &nlabel, bool first)
{

```

```

bool    skip_sort=true;
int     i=0, j=0;
int     crosses, prev_crosses, pprev_crosses, orig_crosses;
char     headline[200];

node_array<int>    x_new(G);
node_array<int>    x_old(G);
array<int>         L_crosses(max_level);
array<int>         L_crosses_new(max_level);
array<int>         L_crosses_old(max_level);

prev_crosses= orig_crosses= crosses=
    all_crossings(G, Level, x_coord, L_crosses);
pprev_crosses= prev_crosses+1;

if (first)
{ down_up_sort(G, Level, x_coord, L_crosses,
               crosses, nlabel, sort_max, DOWN, 0);

#ifdef _DEBUG_SUGI
    sprintf(headline, "hierarchy after normal sort: %d crossings",
crosses);
    if (!draw_hierarchy(G, Level, x_coord, nlabel, headline, true))
user_exit();
#endif
}

if (crosses==0) return 0;

copy_all_xcoord(G, Level, x_coord, x_old, L_crosses, L_crosses_old);
copy_all_xcoord(G, Level, x_coord, x_new, L_crosses, L_crosses_new);


j=perm_max;
while (j--)
{ if (!skip_sort)
    { down_up_sort(G, Level, x_coord, L_crosses,
                  crosses, nlabel, 1, DOWN, 0);
      copy_all_xcoord(G, Level, x_coord, x_new,
                      L_crosses, L_crosses_new);          // update x_new
    }

    i=0; skip_sort=false;
    while (i < max_level)                                // down loop
    { update_best(G, Level, x_coord, x_new,
                  L_crosses, i, crosses, SORT_LOWER | PERMUTE);
      update_best(G, Level, x_coord, x_new,
                  L_crosses, i, crosses, SORT_LOWER);
      if (crosses==0) return 0;
      i++;
    }

#ifdef _DEBUG_SUGI
    sprintf(headline, "hierarchy after down-permute, pass %d: %d
crossings",
perm_max-j, crosses);
    if (!draw_hierarchy(G, Level, x_coord, nlabel, headline, true))
user_exit();
#endif
}

```

```

    i= max_level-1;
    while (i>=0) // up loop
    { update_best(G, Level, x_coord, x_new,
        L_crosses, i, crosses, SORT_UPPER | PERMUTE);
      update_best(G, Level, x_coord, x_new,
        L_crosses, i, crosses, SORT_UPPER);
      if (crosses==0) return 0;
      i--;
    }
#ifdef _DEBUG_SUGI
    sprintf(headline, "hierarchy after up-permute, pass %d: %d
crossings",
    perm_max-j, crosses);
    if (!draw_hierarchy(G, Level, x_coord, nlabel, headline, true))
user_exit();
#endif

    if ((pprev_crosses== crosses) && (prev_crosses== crosses))
        break;

    pprev_crosses= prev_crosses;  prev_crosses= crosses;
} // outer loop


// do_debug=true;

down_up_sort(G, Level, x_coord, L_crosses,
              crosses, nlabel, 2, DOWN, 0);
if (crosses > orig_crosses)
{ copy_all_xcoord(G, Level, x_old, x_coord, L_crosses_old, L_crosses);
  return orig_crosses;
}

return crosses;
}


int improve_coord(graph &G, node_array<int> &the_level, array<list<node>
>
                &Level, node_array<int> &x_coord, node_array<int>
&x_prio,
                node_array<int>& /*nlabel */)
{
    int    i, x_min=100, x_max=-100, width=0, wl=0;
    int    xv, xw, cpos, lpos, rpos, y_prev;
    node    v, w;
    edge    e;
    list_item    lit;
    node_array<list_item>    LIT(G);

    G.sort_nodes(x_coord);

```

```

i=0;
while (i <= max_level) Level[i++].clear();
forall_nodes(v, G) LIT[v]=Level[ the_level[v] ].append(v);

i=0;
while (i < max_level)
    two_level(G, Level, i++, x_coord, x_prio, SORT_LOWER | IMPROVE);

#ifdef _DEBUG_SUGI
    if (!draw_hierarchy(G, Level, x_coord, nlabel, "1st down improving",
true))
        user_exit();
#endif

while (i > 0)
    two_level(G, Level, --i, x_coord, x_prio, SORT_UPPER | IMPROVE);

#ifdef _DEBUG_SUGI
    if (!draw_hierarchy(G, Level, x_coord, nlabel, "1st up improving",
true))
        user_exit();
#endif

i=-1;
while ((++i) <= max_level)
{ if (Level[i].size() > width)
    { width=Level[i].size(); wl=i; }
}

while (wl < max_level)
    two_level(G, Level, wl++, x_coord, x_prio, SORT_LOWER | IMPROVE);

#ifdef _DEBUG_SUGI
    if (!draw_hierarchy(G, Level, x_coord, nlabel, "2nd down improving",
true))
        user_exit();
#endif

forall_nodes(v, G)
{ cpos= lpos= rpos= 0;

    forall_out_edges(e, v)
    { w=G.target(e);
      xv=x_coord[v]; xw=x_coord[w];
      if (xv == xw) cpos++;
      if (xv+1 == xw) rpos++;
      if (xv-1 == xw) lpos++;
    }

    forall_in_edges(e, v)
    { w=G.source(e);
      xv=x_coord[v]; xw=x_coord[w];
      if (xv == xw) cpos++;
      if (xv+1 == xw) rpos++;
    }
}

```



```

    if (xv-1 == xw) lpos++;
}

if ((cpos>=rpos) && (cpes>=lpos)) continue;
lit=LIT[v]; i=the_level[v];
if (lpos> rpos)
{ lit=Level[i].pred(lit);
  if (lit) xw=x_coord[ Level[i].inf(lit) ];
  if (!lit || (xw != xv-1)) x_coord[v]--;
}
else
{ lit=Level[i].succ(lit);
  if (lit) xw=x_coord[ Level[i].inf(lit) ];
  if (!lit || (xw != xv+1)) x_coord[v]++;
}
}
}

```

```

G.sort_nodes(x_coord);
i=0; y_prev=x_coord[G.first_node()];

forall_nodes(v, G)
  if (x_coord[v] == y_prev)
  { y_prev= x_coord[v]; x_coord[v]=i; }
  else
  { y_prev= x_coord[v]; x_coord[v]=++i; }

return i+1;
}

```

```

int SUGIYAMA_and_info(graph &G, node_array<int> &x_coord,
                     node_array<int> &the_level, list<node>
&dummy_nodes,
                     bool first= true)
{
  if (G.number_of_nodes() == 0) // paranoia setting
    return 0;

  int crosses;
  node v;

  max_level= 0;
  forall_nodes(v, G)
    if (the_level[v] > max_level) max_level= the_level[v];

  array<list<node> > Level(max_level+1);

  if (make_hierarchy(G, the_level, Level, dummy_nodes, first) )
    return -1;
  if (first) x_coord.init(G);
}

```

```

node_array<int>  nlabel(G);
node_array<int>  x_prio(G, 0);

forall(v, dummy_nodes)  x_prio[v]= -1;
init_positions(G, Level, x_coord, nlabel, x_prio, first);

#ifdef _DEBUG_SUGI
int    i;
char   head[200];

    i=crosses=0;
    while(i < max_level)
        crosses+= number_of_crossings(G, Level, i++, x_coord);
    sprintf(head, "original hierarchy: %d crossings", crosses);
    if (!draw_hierarchy(G, Level, x_coord, nlabel, head, true))
        user_exit();
#endif

    crosses= down_up_change(G, Level, x_coord, nlabel, first);

#ifdef _DEBUG_SUGI
    sprintf(head, "Sort/permute finished with %d crossings", crosses);
    if (!draw_hierarchy(G, Level, x_coord, nlabel, head, true))
        user_exit();
#endif

    int width= improve_coord(G, the_level, Level, x_coord, x_prio, nlabel);

#ifdef _DEBUG_SUGI
    if (!draw_hierarchy(G, Level, x_coord, nlabel, "final embedding",
        true))
        user_exit();
#endif

    return crosses;
}

int SUGIYAMA_and_info(graph &G, node_array<int> &x_coord,
                        node_array<int> &the_level, list<node>
&dummy_nodes,
                        bool first, char *headline, file_ostream
&ps_out)
{
    if (G.number_of_nodes() == 0)                // paranoia setting
        return 0;

    int  crosses;
    node v;

    max_level= 0;
    forall_nodes(v, G)
        if (the_level[v] > max_level)  max_level= the_level[v];

```

```

array<list<node> > Level(max_level+1);

if (make_hierarchy(G, the_level, Level, dummy_nodes, first) )
    return -1;
if (first) x_coord.init(G);

node_array<int> nlabel(G);
node_array<int> x_prio(G, 0);

forall(v, dummy_nodes) x_prio[v]= -1;
init_positions(G, Level, x_coord, nlabel, x_prio, first);

#ifdef _DEBUG_SUGI
int i;
char head[200];

i=crosses=0;
while(i < max_level)
    crosses+= number_of_crossings(G, Level, i++, x_coord);
sprintf(head, "original hierarchy: %d crossings", crosses);
if (!draw_hierarchy(G, Level, x_coord, nlabel, head, true))
user_exit();
#endif

crosses= down_up_change(G, Level, x_coord, nlabel, first);

#ifdef _DEBUG_SUGI
sprintf(head, "Sort/permute finished with %d crossings", crosses);
if (!draw_hierarchy(G, Level, x_coord, nlabel, head, true))
user_exit();
#endif

int width= improve_coord(G, the_level, Level, x_coord, x_prio, nlabel);

#ifdef _DEBUG_SUGI
if (!draw_hierarchy(G, Level, x_coord, nlabel, "final embedding",
true))
    user_exit();
#endif

write_to_ps(G, Level, x_coord,
            nlabel, width, max_level,
            headline, ps_out);

return crosses;
}

int SUGIYAMA_EMBED(graph &G, node_array<int> &x_coord,
                  node_array<int> &the_level, list<node>
&dummy_nodes)
{
    return SUGIYAMA_and_info(G, x_coord, the_level, dummy_nodes, true);
}

```

```

}

int SUGIYAMA_iterate(graph &G, node_array<int> &x_coord,
                    node_array<int> &the_level, list<node>
&dummy_nodes)
{
    return SUGIYAMA_and_info(G, x_coord, the_level, dummy_nodes, false);
}

int SUGIYAMA_simple(const graph &G, const node_array<int> &the_level)
{
    if (G.number_of_nodes() == 0)                // paranoia setting
        return 0;

    char    filename[200];
    node    v;
    edge    e;
    list<node>    dummy_nodes;
    graph    CG;
    node_array<node>    Cnode(G);

    forall_nodes(v, G) Cnode[v]=CG.new_node();
    forall_edges(e, G) CG.new_edge(Cnode[G.source(e)], Cnode[G.target(e)]
);

    node_array<int>    x_coord(CG);
    node_array<int>    CLevel(CG);
    forall_nodes(v, G) CLevel[Cnode[v]]=the_level[v];

    sprintf(filename, "sugi_out.ps");
    file_ostream    ps_out(filename);

    return SUGIYAMA_and_info(CG, x_coord, CLevel, dummy_nodes, true,
                            filename, ps_out);
}

int SUGIYAMA_EMBEDDING(graph &G, node_array<int>& xcoord,
                    node_array<int>& level,
                    edge_array<list<int>> &xpoly)
{
    list<node>    dummy_nodes;

    int crossings = SUGIYAMA_and_info(G, xcoord, level, dummy_nodes, true);

    xpoly.init(G);

    node_array<bool>    dummy(G, false);

    node v;
    forall(v, dummy_nodes) dummy[v] = true;

```

```

forall_nodes(v,G)
{ if (dummy[v]) continue;
  edge e = G.first_adj_edge(v);
  while (e)
  { edge next = G.adj_succ(e);
    edge x = e;
    node u = target(x);
    while (dummy[u])
    { xpoly[e].append(xcoord[u]);
      x = G.first_adj_edge(u);
      u = target(x);
    }
    if (u != target(e))
      G.move_edge(e,source(e),u);
    e = next;
  }
}

forall(v,dummy_nodes) G.del_node(v);

return crossings;
}

```

```

/*****
+
+ LEDA 3.5.1
+
+ _tutte.c
+
+ This file is part of the LEDA research version (LEDA-R) that can be
+ used free of charge in academic research and teaching. Any commercial
+ use of this software requires a license which is distributed by the
+ LEDA Software GmbH, Postfach 151101, 66041 Saarbruecken, FRG
+ (fax +49 681 31104).
+
+ Copyright (c) 1991-1997 by Max-Planck-Institut fuer Informatik
+ Im Stadtwald, 66123 Saarbruecken, Germany
+ All rights reserved.
+
*****/
// ----- //
// drawing of a graph using Tutte's algorithm //
// David Alberts (1996) //
// ----- //

#include <LEDA/graph_alg.h>
#include <LEDA/vector.h>
#include <LEDA/matrix.h>

bool TUTTE_EMBEDDING(const graph& G, const list<node>& fixed_nodes,
                     node_array<double>& xpos, node_array<double>& ypos)
{
    // computes a convex drawing of the graph G if possible. The list
    // fixed_nodes contains nodes with prescribed coordinates already
    // given in pos. The computed node positions of the other nodes are
    // stored in pos, too. If the operation is successful, true is
    // returned.
    // Precondition: pos is valid for G.

    node v,w;
    edge e;
    node_array<bool> is_fixed(G,false);
    forall(v,fixed_nodes) is_fixed[v] = true;

    list<node> other_nodes;
    forall_nodes(v,G) if(!is_fixed[v]) other_nodes.append(v);
    node_array<int> ind(G); // position of v in other_nodes and A
    int i = 0;
    forall(v,other_nodes) ind[v] = i++;

    int n = other_nodes.size(); // #other nodes
    vector coord(n); // coordinates (first x then y)
    vector rhs(n); // right hand side
    matrix A(n,n); // equations

    // initialize non-zero entries in matrix A
    forall(v,other_nodes)
    {
        double one_over_d = 1.0/double(G.degree(v));
        forall_inout_edges(e,v)
        {

```

```

        // get second node of e
        w = (v == source(e)) ? target(e) : source(e);
        if(!is_fixed[w]) A(ind[v],ind[w]) = one_over_d;
    }
    A(ind[v],ind[v]) = -1;
}

if(!A.det()) return false;

// compute right hand side for x coordinates
forall(v,other_nodes)
{
    rhs[ind[v]] = 0;
    double one_over_d = 1.0/double(G.degree(v));
    forall_inout_edges(e,v)
    {
        // get second node of e
        w = (v == source(e)) ? target(e) : source(e);
        if(is_fixed[w]) rhs[ind[v]] -= (one_over_d*xpos[w]);
    }
}

// compute x coordinates
coord = A.solve(rhs);
forall(v,other_nodes) xpos[v] = coord[ind[v]];

// compute right hand side for y coordinates
forall(v,other_nodes)
{
    rhs[ind[v]] = 0;
    double one_over_d = 1.0/double(G.degree(v));
    forall_inout_edges(e,v)
    {
        // get second node of e
        w = (v == source(e)) ? target(e) : source(e);
        if(is_fixed[w]) rhs[ind[v]] -= (one_over_d*ypos[w]);
    }
}

// compute y coordinates
coord = A.solve(rhs);
forall(v,other_nodes) ypos[v] = coord[ind[v]];

return true;
}

```


APPENDIX B: CODE

A. TEST CODE CREATED

```
#include "graph_layout.h"

//
// MAIN PROGRAM
//
main()
{
    GRAPH<int,int> G;

    init_graph(G, true);

    run_ortho(G, true);
    run_straight_line(G, true);
    run_straight_line2(G, true);
    run_tutte(G, true);
    run_spring_embedding(G, true);
    run_d2_spring_embedding(G, true);

    return 0;
}
```

B. NEW CODE CREATED

```
#include <LEDA/graph.h>
#include <LEDA/graph_alg.h>

#ifdef GRAPH_LAYOUT_H
#define GRAPH_LAYOUT_H 1

void run_ortho(graph&, bool);
void run_straight_line(graph&, bool);
void run_straight_line2(graph&, bool);
void run_tutte(graph&, bool);
void run_d2_spring_embedding(graph&, bool);
void run_spring_embedding(graph&, bool);
void init_graph(graph&, bool);

#endif
```

```

#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>
#include <stream.h>
#include "graph_layout.h"

//
//      THIS IS THE ORTHOGONAL LAYOUT ALGO
//
void run_ortho(graph& G, bool DEBUG)
{
    node v;
    node_array<int> x(G),y(G);
    edge_array< list<int> > xbends(G), ybends(G);

    if (DEBUG)
    {
        cout << "NO EMBEDDING      ";
        newline;
    }

    forall_nodes(v,G)
    {
        x[v] = rand_int(1,500);
        y[v] = rand_int(1,500);
    }

    if (DEBUG)
    {
        forall_nodes(v,G) cout << string("x = %3d      y = %3d\n",x[v],y[v]);
        cout << "ORTHO EMBEDDING      ";
        newline;
    }

    ORTHO_EMBEDDING(G, x, y, xbends, ybends, false);

    if (DEBUG)
    {
        forall_nodes(v,G) cout << string("x = %3d      y = %3d\n",x[v],y[v]);
        G.write_gml("graph.ortho");
    }

}

//
//      THIS IS THE STRAIGHT_LINE_2 ALGO
//
void run_straight_line2(graph& G, bool DEBUG)
{
    node v;
    node_array<int> x(G),y(G);

    if (DEBUG)
    {
        cout << "NO EMBEDDING      ";
        newline;
    }

    forall_nodes(v,G)
    {
        x[v] = rand_int(1,500);

```

```

    y[v] = rand_int(1,500);
}
if (DEBUG)
{
    forall_nodes(v,G) cout << string("x = %3d    y = %3d\n",x[v],y[v]);
    cout << "STRAIGHT LINE EMBEDDING2    ";
    newline;
}

STRAIGHT_LINE_EMBEDDING2(G, x, y);

if (DEBUG)
{
    forall_nodes(v,G) cout << string("x = %3d    y = %3d\n",x[v],y[v]);
    G.write_gml("graph.straight_12");
}

}

//
//      THIS IS THE STRAIGHT LINE ALGO
//
void run_straight_line(graph& G, bool DEBUG)
{
    node v;
    node_array<int> x(G), y(G);

    if (DEBUG)
    {
        cout << "NO EMBEDDING    ";
        newline;
    }

    forall_nodes(v,G)
    {
        x[v] = rand_int(1,500);
        y[v] = rand_int(1,500);
    }
    if (DEBUG)
    {
        forall_nodes(v,G) cout << string("x = %3d    y = %3d\n",x[v],y[v]);
        cout << "STRAIGHT LINE EMBEDDING    ";
        newline;
    }

    STRAIGHT_LINE_EMBEDDING(G, x, y);

    if (DEBUG)
    {
        forall_nodes(v,G) cout << string("x = %3d    y = %3d\n",x[v],y[v]);
        G.write_gml("graph.straight_1");
    }

}

//
//      THIS IS THE TUTTE ALGO
//
void run_tutte(graph& G, bool DEBUG)
{
    node_array<double> dx(G), dy(G);

```

```

list<node> L;
node v;

if (DEBUG)
{
cout << "NO EMBEDDING          ";
newline;
}

forall_nodes(v,G)
{
    dx[v] = (double) rand_int(1,500);
    dy[v] = (double) rand_int(1,500);
}
if (DEBUG)
{
forall_nodes(v,G) cout << string("x = %lf      y = %lf\n",dx[v],dy[v]);
cout << "TUTTE EMBEDDING          ";
newline;
}

TUTTE_EMBEDDING(G,L,dx,dy);

if (DEBUG)
{
forall_nodes(v,G) cout << string("x = %lf      y = %lf\n",dx[v],dy[v]);
G.write_gml("graph.tutte");
}

}

//
//      THIS IS THE D2 SPRING EMBEDDING ALGO
//
void run_d2_spring_embedding(graph& G, bool DEBUG)
{
node_array<double> dx(G), dy(G);
node v;

if (DEBUG)
{
cout << "NO EMBEDDING          ";
newline;
}

forall_nodes(v,G)
{
    dx[v] = (double) rand_int(1,500);
    dy[v] = (double) rand_int(1,500);
}
if (DEBUG)
{
forall_nodes(v,G) cout << string("x = %lf      y = %lf\n",dx[v],dy[v]);
cout << "D2 SPRING EMBEDDING EMBEDDING          ";
newline;
}

D2_SPRING_EMBEDDING(G,dx,dy,0.0,500.0,0.0,500.0,400);

```

```

if (DEBUG)
{
forall_nodes(v,G) cout << string("x = %lf    y = %lf\n",dx[v],dy[v]);
G.write_gml("graph.d2_spr_emb");
}

}

//
//      THIS IS THE SPRING EMBEDDING ALGO
//
void run_spring_embedding(graph& G, bool DEBUG)
{
node_array<double> dx(G), dy(G);
node v;

if (DEBUG)
{
cout << "NO EMBEDDING      ";
newline;
}

forall_nodes(v,G)
{
dx[v] = (double) rand_int(1,500);
dy[v] = (double) rand_int(1,500);
}
if (DEBUG)
{
forall_nodes(v,G) cout << string("x = %lf    y = %lf\n",dx[v],dy[v]);
cout << "SPRING EMBEDDING EMBEDDING      ";
newline;
}

SPRING_EMBEDDING(G,dx,dy,0.0,500.0,0.0,500.0,400);

if (DEBUG)
{
forall_nodes(v,G) cout << string("x = %lf    y = %lf\n",dx[v],dy[v]);
G.write_gml("graph.spr_emb");
}

}

//
//      INITIALIZE GRAPH TO BIDIRECTIONAL AND PLANAR
//
void init_graph(graph& G, bool DEBUG)
{
if (DEBUG)
{
cout << "Inside init_graph.";
newline;
}
int n = read_int("n = ");
random_planar_graph(G,n,n*2);

```

```
list<edge> bi_edges = Make_Bidirected(G);  
G.make_map();  
G.make_planar_map();  
  
}
```

C. CAPS CODE MODIFIED

/*

NOTES:

*/

/* *****

Name: graph_editor.C
Author: Capt Robert M. Dixon
Program: graph_editor
Date Modified: 21 Sep 92
Remarks: graph_editor.C is the main program for the
CAPS '93 graph editor. Depending on the command line
parameters, it allows either viewing only or full
editing of a graph passed by the CAPS '93 syntax-
directed editor.

General Comments:

The XmProcessTraversal function is called numerous
places in an attempt to keep the keyboard input focus
in the drawing window. This allows the editor to
respond to the delete and backspace key. This works
with varying degrees of success.

Credits: Portions of code are adapted from the following:
Barakati, Naba, X Window System Programming, SAMS,
1991.

Heller, Dan, Motif Programming Manual, O'Reilly and
Associates, 1991.

Johnson, Eric, and Reichard, Kevin, X Window
Applications Programming, MIS Press, 1989.

Young, Douglas, Object Oriented Programming With C++
and OSF/Motif, Prentice-Hall, 1992.

Reengineering:

Modified by Doug Lange on 8/12/96

Removed Property button and put in its place a Timers

button

Modified by Doug Lange on 8/16/96 - 8/20/96

Added callback and dialog for Timer Tool button.

Modified by Doug Lange on 8/19/96

Added callback and dialog for Informal Description Tool

button.

History:

@1 96/09/29 Ken Moeller

Migration from Motif 1.2 to Motif 1.1.

- @2 96/10/01 Ken Moeller
Upgraded calling arguments to reflect changes to requirements.
- @3 96/10/03 Ken Moeller
Started to switch over to build_from_sde and write_to_sde.
This is not yet complete.
- @4 96/10/03 Ken Moeller
Items removed or tests added while testing @3.
- @5 96/10/04 Ken Moeller
Removal of viewer code. This option is no longer supported.
Still need to investigate the resources. So the job is not
complete.
- @6 96/10/06 Ken Moeller
Change in how units are encoded.
- @7 96/10/11 Ken Moeller
Moved print command over to an XEvent so that the window
can be refreshed before the screen is captured.

***** */

```
#include <fstream.h>           //Added by DL 8/19/96
#include <iostream.h>          //Added by DL 8/19/96
#include <stdlib.h>
#include <stream.h>
#include <sys/stat.h>           //Added by DHA 9/18/96
#include <sys/types.h>          //Added by DHA 9/18/96
```

```
#include <X11/Xatom.h>
#include <X11/cursorfont.h>
#include <X11/keysym.h>
#include <Xm/DialogS.h>         //Added by DL 8/19/96
#include <Xm/DrawingA.h>
#include <Xm/DrawnB.h>
#include <Xm/Form.h>
#include <Xm/LabelG.h>
#include <Xm/List.h>
#include <Xm/MainW.h>
#include <Xm/MessageB.h>
#include <Xm/PanedW.h>          //Added by DL 8/19/96
#include <Xm/PushB.h>           //Added by DL 8/16/96
#include <Xm/PushBG.h>          //Added by DL 8/19/96
#include <Xm/RowColumn.h>
#include <Xm/ScrolledW.h>
#include <Xm/SelectioB.h>
#include <Xm/Separator.h>       //Added by DHA 8/20/96
#include <Xm/Text.h>
#include <Xm/TextF.h>
#include <Xm/ToggleBG.h>
```

```
//#include "ge_utilities_debug.h"
#include <stdio.h>
#include <math.h>
```

```
#include "action_area.h"        //Added by DL 8/19/96
#include "build_option.h"       //Added by DHA 8/20/96
```

```

#include "get_unique_id.h"           //Added by MY 8/4/97
#include "ge_defs.h"                 //Added by DL 8/16/96
#include "ge_interface.h"            //Added by DL 8/16/96
#include "gettopshell.h"             //Added by DHA 8/19/96
#include "graph_editor.h"            // kbm
#include "graph_object_list.h"
//#include "op_prop_exception.h"     //Added for req. 7.4, dha
9/12/96
//#include "op_prop_formal_desc.h"   //Added for req. 7.6, dha
9/15/96
//#include "op_prop_informal_desc.h" //Added for req. 7 dha 9/15/96
//#include "op_prop_keywords.h"      //Added for req. 7.6, dha
9/15/96
//#include "op_prop_output_guard.h"   //Added for req. 7.3, dha
9/12/96
//#include "op_prop_timer_op.h"      //Added for req. 7.5, dha
9/12/96
//#include "op_prop_timing_info.h"   //Added for req. 7 & 7.7, dha
9/5/96
//#include "op_prop_trigger_cond.h"   //Added for req. 7.2, dha
9/12/96
#include "operator_object.h"
#include "postpopup.h"               //Added by DHA 8/20/96
#include "setcursor.h"               //Added by DHA 8/15/96
#include "spline_object.h"
#include "stream_object.h"
#include "operator_property_menu.h"
#include "stream_property_menu.h"    //Added for req. 8, dha 9/16/96
#include "timer_tool.h"              //Added by DL 8/22/96
#include "windows.h"                 // kbm
#include "warning.h"
#include "ge_utilities_debug.h"
#include "report_errors.h"
#include "graph_layout.h"            // Added by BR 9/6/97

// MAXCOLORS is the number of colors defined to the editor.
// To add or subtract colors, this value must be modified.

// changed size from 75
#define BUTTONWIDTH 65
#define HELPSIZ 1000

// graph_editor has a number of global variables due to
// Motif's use of callback functions. Since these functions
// have fixed formal parameter lists, global variables must
// be used to pass some data between functions.

// All drawing commands are executed on both drawing_a and
// drawing_area_pixmap. drawing_a is the visible canvas, while
// drawing_area_pixmap provides a backup. When the canvas needs
// to be redrawn, the drawing in drawing_area pixmap is merely
// copied back onto the canvas.

// colors[] is a list of predefined X colors. To use others,
// consult an X reference giving allowable color names. Using
// the predefined colors allows the user to specify color
// preferences in X resource text files.

// graphic_list is a GraphObjectList containing all the
// visible operators and streams.

```

```

//  selected_object_ptr always points to the object selected
//  (i. e. with handles around it) on the drawing canvas.

//  num_del_ops is the number of deleted operators, and
//  del_op_id is an array of identifiers for deleted operators.

//  The Resrcs struct, resources[], and options[] are used
//  by Motif for parsing the command line options.

Widget toplevel, main_w, menubar, rowcol, scrolled_win,
      op_button, term_button, stream_button, select_button,
      spec_button, informal_button, types_button,
      timers_button, button_divider;
XtAppContext app;
Pixmap op_button_pixmap, term_button_pixmap, stream_button_pixmap,
      select_button_pixmap, spec_button_pixmap,
informal_button_pixmap,
      types_button_pixmap,
      timers_button_pixmap;
XGCValues gcv1, gcv2, gcv3;
Screen *screen_ptr;
XtActionsRec actions;
String translations =
    "<Btn1Down>:  draw(down)\n\
    <Btn1Up>:    draw(up)  \n\
    <Btn1Motion>: draw(motion)\n\
    <Btn3Down>:  draw(btn3down)\n\
    <Btn3Motion>: draw(btn3motion)\n\
    <Btn3Up>:    draw(btn3up)\n\
    <MotionNotify>: draw(motionnotify)\n\
    <Key>:       draw(key)\n\
    <Key>Tab:    draw(tab)";
unsigned long gc_mask;
Window root_window, toplevel_window;
//  XEvent *print_event = (XEvent *) malloc(sizeof(XEvent));    // @7
XEvent *print_event;

extern int    Global_argc;
extern char **Global_argv;

// MY 8/5/97
extern int enter_operator;
extern int enter_stream;
extern int enter_errs;
int enter_types;
int enter_spec;
int enter_timer;
int enter_inform;

// MY 8/5/97
void types_close_dialog(Widget w, XtPointer client_data, XtPointer
call_data) {
    enter_types = 0;
    close_dialog(w, client_data, call_data);
}

void spec_close_dialog(Widget w, XtPointer client_data, XtPointer
call_data) {
    enter_spec = 0;
    close_dialog(w, client_data, call_data);
}

```

```

}

void timer_close_dialog(Widget w, XtPointer client_data, XtPointer
call_data) {
    enter_timer = 0;
    close_dialog(w, client_data, call_data);
}

void inform_close_dialog(Widget w, XtPointer client_data, XtPointer
call_data) {
    enter_inform = 0;
    close_dialog(w, client_data, call_data);
}

int still_open()
{
    return ( enter_types || enter_spec || enter_timer || enter_inform ||
            enter_operator || enter_stream || enter_errs );
}

int is_empty(char * str)
{
    if ( *str == ' ' ) return 1;
    return 0;
}

/*****st
ream
*** Added by Doug Lange 8/16/96.*/
GRAPH_DESC gdnodes;
ID_LIST idp;

ACTION_NODE* next_action_ptr;          // kbm

GC std_graphics_context, dotted_context, erase_context;
Dimension width, height;
Pixmap drawing_area_pixmap;
Widget drawing_a, current_op_name, current_op_met;
Widget save_indicator, error_indicator, status_indicator;
BOOLEAN state_stream = false, alt_selected = false, ctrl_selected =
false;
BOOLEAN ibar_mode = false; // added for req #6.1. dha
BOOLEAN label_edit_mode = false; // added for req #6.1.1. dha
//MY 8/4/97
char default_name[INPUT_LINE_SIZE]; // added for req #6.4 dha
CLASS_DEF object_def = GRAPHOBJECT; // added for req #6.1. dha
char* colors[] = {"Aquamarine", "Black", "Blue", "BlueViolet",
                  "Brown", "CadetBlue", "Coral",
                  "CornflowerBlue", "Cyan", "DarkGreen",
                  "DarkOliveGreen", "DarkOrchid",
                  "DarkSlateBlue", "DarkSlateGrey",
                  "DarkTurquoise", "DimGrey", "Firebrick",
                  "ForestGreen", "Gold", "Goldenrod", "Grey",
                  "Green", "GreenYellow", "IndianRed", "Khaki",
                  "LightBlue", "LightGrey", "LightSteelBlue",
                  "LimeGreen", "Magenta", "Maroon",
                  "MediumAquamarine", "MediumBlue",
                  "MediumOrchid", "MediumSeaGreen",
                  "MediumSlateBlue", "MediumSpringGreen",
                  "MediumTurquoise", "MediumVioletRed",
                  "MidnightBlue", "Navy", "Orange", "OrangeRed",

```

```

        "Orchid", "PaleGreen", "Pink", "Plum", "Red",
        "Salmon", "SeaGreen", "Sienna", "SkyBlue",
        "SlateBlue", "SpringGreen", "SteelBlue",
        "Tan", "Thistle", "Turquoise", "Violet",
        "VioletRed", "Wheat", "White", "Yellow",
        "YellowGreen");
unsigned long color_table[MAXCOLORS + 1];
TOOL_STATE tool_state = SELECT_TOOL;
GraphObjectList graphic_list;
GraphObject* selected_object_ptr = NULL;
OperatorObject *op_being_updated = NULL; // Add for req. 7, dha
StreamObject *st_being_updated = NULL; // Add for req. 8, dha
Display *display_ptr;
Window draw_window;
int default_color = WHITE;
int default_font = COURIERBOLD12;
int num_del_ops = 0;
OP_ID del_op_id[MAXDELETEDOPS];
ERROR_MSGS errors_present;
BOOLEAN psdl_modified, syntax_checked;
int save_performed; // updated save_state when you return
char *help_menu_files[] = {"psdl_grammar.hlp",
                           "operators.hlp",
                           "streams.hlp",
                           "exceptions.hlp",
                           "timers.hlp"};

// ?? Look at this to see if still needed *****
struct _resrcs {
    int viewer;
} Resrcs;

static XtResource resources[] = {
    {"viewer", "Viewer", XmRBoolean, sizeof (int),
     XtOffsetOf(struct _resrcs, viewer), XmRImmediate, False},
};

static XrmOptionDescRec options[] = {
    {"-v", "viewer", XrmoptionNoArg, "True"},
};

//*****

void select_state(TOOL_STATE new_state) {
    tool_state = new_state;

    if (new_state == OPERATOR_TOOL)
        XtVaSetValues(op_button, XmNshadowType, XmSHADOW_IN, NULL);
    else
        XtVaSetValues(op_button, XmNshadowType, XmSHADOW_OUT, NULL);

    if (new_state == TERMINATOR_TOOL)
        XtVaSetValues(term_button, XmNshadowType, XmSHADOW_IN, NULL);
    else
        XtVaSetValues(term_button, XmNshadowType, XmSHADOW_OUT, NULL);

    if (new_state == STREAM_TOOL)
        XtVaSetValues(stream_button, XmNshadowType, XmSHADOW_IN, NULL);
    else
        XtVaSetValues(stream_button, XmNshadowType, XmSHADOW_OUT, NULL);
}

```

```

    if (new_state == SELECT_TOOL)
        XtVaSetValues(select_button, XmNshadowType, XmSHADOW_IN, NULL);
    else
        XtVaSetValues(select_button, XmNshadowType, XmSHADOW_OUT, NULL);

    // Display all other buttons
    /* ?? delete if not needed
    XtVaSetValues(types_button, XmNshadowType, XmSHADOW_OUT, NULL);
    XtVaSetValues(spec_button, XmNshadowType, XmSHADOW_OUT, NULL);
    XtVaSetValues(timers_button, XmNshadowType, XmSHADOW_OUT, NULL);
    XtVaSetValues(informal_button, XmNshadowType, XmSHADOW_OUT, NULL);
    */
}

/*****
 * error_label() --
*****/
void error_label() {
    XmString label;

    /*
    * MY 7/22/97
    */

    if ((errors_present == NULL) || (!syntax_checked)) {
        label = XmStringCreateSimple(" Check "); // MY: Check (Syntax)
        XtVaSetValues(error_indicator, XmNlabelString, label, NULL);
        XtVaSetValues(error_indicator, XmNshadowType, XmSHADOW_OUT, NULL);
    }
    else {
        label = XmStringCreateSimple("ERROR MSGS");
        XtVaSetValues(error_indicator, XmNlabelString, label, NULL);
        XtVaSetValues(error_indicator, XmNshadowType, XmSHADOW_OUT, NULL);
    }

    XmStringFree(label);
    /*
    * unmasked 8/6/97
    */
}

/*****
 * save_state() -- Updates the save_indicator with the current indicated
 * state.
*****/
void save_state(int state) {
    XmString label;

    if (state == NOT_MODIFIED) {
        label = XmStringCreateSimple("Save Not Required");
    }
}

```

```

    XtVaSetValues(save_indicator, XmNlabelString, label, NULL);
    XtVaSetValues(save_indicator, XmNshadowType, XmSHADOW_IN, NULL);
    psdl_modified = false;
}
else if (state == SAVE_REQUIRED) {
    label = XmStringCreateSimple("SAVE REQUIRED");
    XtVaSetValues(save_indicator, XmNlabelString, label, NULL);
    XtVaSetValues(save_indicator, XmNshadowType, XmSHADOW_OUT, NULL);
    psdl_modified = true;
    syntax_checked = false;
}
else {
    label = XmStringCreateSimple("");
    XtVaSetValues(save_indicator, XmNlabelString, label, NULL);
    XtVaSetValues(save_indicator, XmNshadowType, XmSHADOW_IN, NULL);
}
XmStringFree(label);

error_label();
}

void update_status(char *status, BOOLEAN bell) {
    XtVaSetValues(status_indicator, XmNvalue, status, NULL);
    if (bell)
        XBell(display_ptr, 100);
}

void clear_status() {
    XtVaSetValues(status_indicator, XmNvalue, "", NULL);
}

//  Initializes the color table.

void initialize_color_table(Screen *screen) {
    Colormap color_map = DefaultColormapOfScreen(screen);
    XColor color, unused;
    int i, screen_depth = DefaultDepthOfScreen(screen);

    if (screen_depth > 1) {          //  a color screen
        for(i = 1; i <= MAXCOLORS; i++) {
            if (!XAllocNamedColor(display_ptr, color_map,
                colors[i - 1], &color, &unused))
                printf ("Allocated unknown color: %s\n", colors[i-1]);
            color_table[i] = color.pixel;
        }
    }
    else {                          //  a black and white screen
        for(i = 1; i <= MAXCOLORS; i++) {
            if (strcmp(colors[i - 1], "White") != 0)
                color_table[i] = BlackPixelOfScreen(screen);
            else
                color_table[i] = WhitePixelOfScreen(screen);
        }
    }
}

/*****
****

```

```

*   Executes menu options from the 'file' menu. This is
*   called by either the menu callback function, if the
*   pulldown menus are used, or by the draw() function,
*   if the alt-key combinations are used.
*****
**/

void handle_file_options(int item_no) {

    int action;

    Quest_Script abort_script =
        {"", "Abort changes made to graph?", "Yes", "No", "Cancel",
BTN2};
    Quest_Script save_script =
        {"", "Save changes made to graph?", "Yes", "No", "Cancel",
BTN1};

    XFlush(display_ptr);
    switch(item_no) {

        case 0: // Save

            // MY 8/5/97
            if ( still_open() )
            {
                warning(drawing_a, "Please close other windows first");
                break;
            }

            // Check for error condition of no Root...this should not be
possible
            next_action_ptr->option      = SAVE_TO_DISK;
            next_action_ptr->reinvoke    = true;
            free(next_action_ptr->next_op);
            next_action_ptr->next_op     = graphic_list.current_op_name();
            next_action_ptr->next_op_num = graphic_list.current_op_num();
            return_sde_flag              = true;

            break;

        case 1: // Restore from Save

            // MY 8/5/97
            if ( still_open() )
            {
                warning(drawing_a, "Please close other windows first");
                break;
            }

            // Check for error condition of no Root...this should not be
possible
            action = YES; // Default action if not
modified
            if (psdl_modified)
                action = AskUser(app, drawing_a, abort_script);
            switch(action) {
                case YES:
                    next_action_ptr->option      = REVERT;
                    next_action_ptr->reinvoke    = true;
                    free(next_action_ptr->next_op);

```



```

        next_action_ptr->next_op      = graphic_list.current_op_name();
        next_action_ptr->next_op_num  = graphic_list.current_op_num();
        return_sde_flag              = true;
        break;

        case NO:
        return_sde_flag = false;  // Aborted operation, do nothing
        break;
    }

    break;

case 2:  // Print

    // MY 8/5/97
    if ( still_open() )
    {
        warning(drawing_a, "Please close other windows first");
        break;
    }

    AskPrint(app,drawing_a, &PrintCmd);
    if (PrintCmd.answer == OK) {
        XSendEvent(display_ptr, toplevel_window, True, 0, print_event);
    }

    break;

/*
 * MY
 */
case 3:  // Abandon Changes

    // MY 8/5/97
    if ( still_open() )
    {
        warning(drawing_a, "Please close other windows first");
        break;
    }

    Quest_Script abandon_script =
        {"","All changes will be lost, are you sure?",
        "Yes","No","Cancel", BTN1};

    action = AskUser(app, drawing_a, abandon_script);
    if (action == YES) {
        next_action_ptr->option      = ABANDON;
        next_action_ptr->reinvoke    = true;
        free(next_action_ptr->next_op);
        next_action_ptr->next_op     = graphic_list.current_op_name();
        next_action_ptr->next_op_num = graphic_list.current_op_num();
        return_sde_flag             = true;
    }

    break;

case 4:  // Exit

    // MY 8/5/97
    if ( still_open() )
    {

```

```

        warning(drawing_a, "Please close other windows first");
        break;
    }

    action = NO;    // Default action if not modified
                   // This is not the default save option, see
save_script
    if (psdl_modified)
        action = AskUser(app,drawing_a, save_script);

    switch(action) {
        case YES:
            next_action_ptr->option    = SAVE_TO_DISK;
            next_action_ptr->reinvoke = false;
            free(next_action_ptr->next_op);
            next_action_ptr->next_op    = graphic_list.root_op_name();
            next_action_ptr->next_op_num = graphic_list.root_op_num();
            return_sde_flag = true;
            break;

            case NO:
                next_action_ptr->option    = ABANDON;
                next_action_ptr->reinvoke    = false;
                free(next_action_ptr->next_op);
                next_action_ptr->next_op    = graphic_list.root_op_name();
                next_action_ptr->next_op_num = graphic_list.root_op_num();
                return_sde_flag    = true;
                break;

            case CANCEL:
            default:
                return_sde_flag = false;
                break;
        }

        break;

    default:

        break;
    }
}

/*****
*   Executes menu options from the 'psdl' menu. This is
*   called by either the menu callback function, if the
*   pulldown menus are used, or by the draw() function,
*   if the alt-key combinations are used.
*****/

void handle_psdl_options(int item_no) {

    int action;
    char *opName;

    Quest_Script abort_script =
        {"", "Abort changes made to graph?", "Yes", "No", "Cancel",
BTN2};

```

```

    Quest_Script save_script =
        {"", "Save changes made to graph?", "Yes", "No", "Cancel",
BTN1};

    XFlush(display_ptr);
    switch(item_no) {

/*
 * MY
 */
    case 3: // Syntax Check

        // MY 8/5/97
        if ( still_open() )
        {
            warning(drawing_a, "Please close other windows first");
            break;
        }

        next_action_ptr->option      = CHECK_SYNTAX;
        next_action_ptr->reinvoke    = true;
        free(next_action_ptr->next_op);
        next_action_ptr->next_op     = graphic_list.current_op_name();
        next_action_ptr->next_op_num = graphic_list.current_op_num();
        return_sde_flag             = true;

        break;

/*
 * unmasked 8/6/97
 */

    case 0: // Go to Root

        // MY 8/5/97
        if ( still_open() )
        {
            warning(drawing_a, "Please close other windows first");
            break;
        }

        // Check for error condition of no Root...this should not be
        possible
        if (graphic_list.root_op_num() == UNDEFINED_OPNUM) {
            warning(drawing_a, "No Root node defined");

            break;
        }

        next_action_ptr->option      = UPDATE_TREE;
        next_action_ptr->reinvoke    = true;
        free(next_action_ptr->next_op);
        next_action_ptr->next_op     = graphic_list.root_op_name();
        next_action_ptr->next_op_num = graphic_list.root_op_num();
        return_sde_flag             = true;

        break;

    case 1: // Go to Parent

        // MY 8/5/97

```

```

if ( still_open() )
{
    warning(drawing_a, "Please close other windows first");
    break;
}

// Check for error condition of no Parent
if (graphic_list.parent_op_num() == UNDEFINED_OPNUM) {
    warning(drawing_a, "No parent node defined");

    break;
}

next_action_ptr->option      = UPDATE_TREE;
next_action_ptr->reinvoke    = true;
free(next_action_ptr->next_op);
next_action_ptr->next_op     = graphic_list.parent_op_name();
next_action_ptr->next_op_num = graphic_list.parent_op_num();
return_sde_flag = true;

break;

case 2: // Decompose

// MY 8/5/97
if ( still_open() )
{
    warning(drawing_a, "Please close other windows first");
    break;
}

if (selected_object_ptr == NULL)
    warning(drawing_a, "Please select an operator");

else {
    if (selected_object_ptr->is_a() == OPERATOROBJECT) {
        opName = selected_object_ptr->name();
        if (strchr(opName, '.') != NULL) { // Is a type
            warning(drawing_a, "Not allowed to decompose a Type Operator");
            update_status(
                "A Type Operator must be Atomic: rename or leave Atomic",
                RING_BELL);
            free(opName);
        }
        else {
            next_action_ptr->option      = UPDATE_TREE;
            next_action_ptr->reinvoke    = true;
            free(next_action_ptr->next_op);
            next_action_ptr->next_op     = opName;
            next_action_ptr->next_op_num =
                ((OperatorObject *) selected_object_ptr)-
>op_num();
            return_sde_flag = true;
        }
    }
    else
        warning(drawing_a, "Please select an operator");
}

break;

```

```

default:
    return_sde_flag = false;

    break;
}
}

// This function is called when a selection is made from
// the list box displayed in the 'draw_options:Color' menu.

void color_list_cb(Widget widget, XtPointer,
                   XtPointer cb_struct_ptr) {
    XmListCallbackStruct *list_struct_ptr =
        (XmListCallbackStruct *) cb_struct_ptr;

    if (selected_object_ptr != NULL) {
        if (selected_object_ptr->is_a() == OPERATOROBJECT) {
            selected_object_ptr->erase();
            selected_object_ptr->color(list_struct_ptr->item_position);
            selected_object_ptr->draw(SOLID);
            save_state(SAVE_REQUIRED);
        }
    }
    else
        default_color = list_struct_ptr->item_position;
    XtDestroyWidget(widget);
}

// This function is called when a selection is made from
// the list box displayed in the 'draw_options:Font' menu.

void font_list_cb(Widget widget, XtPointer,
                  XtPointer cb_struct_ptr) {
    XmListCallbackStruct *list_struct_ptr =
        (XmListCallbackStruct *) cb_struct_ptr;

    if (selected_object_ptr != NULL) {
        selected_object_ptr->erase();
        selected_object_ptr->set_object_font(list_struct_ptr->
item_position);
        selected_object_ptr->draw(SOLID);
        save_state(SAVE_REQUIRED);
    }
    else {
        default_font = list_struct_ptr->item_position;
        graphic_list.set_default_font(default_font);
    }
    XtDestroyWidget(widget);
    XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);
}

// This function is called when a selection is made from
// the list box displayed in the 'draw_options:Undelete Operator'
// menu.

static void op_list_cb(Widget widget, XtPointer,
                       XtPointer cb_struct_ptr) {
    XmListCallbackStruct *list_struct_ptr =
        (XmListCallbackStruct *) cb_struct_ptr;

```

```

// The last entry in the list is 'Cancel'.
if (list_struct_ptr->item_position != num_del_ops + 1) {

    graphic_list.set_undeleted(OPERATOROBJECT,
                                del_op_id[list_struct_ptr->item_position
- 1]);
    save_state(SAVE_REQUIRED);
    graphic_list.draw();
}
XtDestroyWidget(widget);
XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);
}

// Executes menu options from the 'Edit' menu. This is
// called by either the menu callback function, if the
// pulldown menus are used, or by the draw() function,
// if the alt-key combinations are used.

void handle_edit_options(int item_no) {
    int i, num_items = XtNumber(colors);
    int reply;
    XmStringTable color_list, font_list, op_list;
    Widget list_box, op_box;
    char *del_op_str[MAXDELETEDOPS];

    switch(item_no) {
        case 0:

            // MY 8/5/97
            if ( still_open() )
            {
                warning(drawing_a, "Please close other windows first");
                break;
            }

            color_list =
                (XmStringTable) XtMalloc(num_items * sizeof(XmString *));
            for(i = 0; i < num_items; i++)
                color_list[i] = XmStringCreateSimple(colors[i]);
            list_box =
                XmCreateScrolledList(drawing_a, "Colors", NULL, 0);
            XtVaSetValues(list_box,
                XmNitems, color_list,
                XmNitemCount, num_items,
                XmNvisibleItemCount, 8,
                NULL);
            for(i = 0; i < num_items; i++)
                XmStringFree(color_list[i]);
            XtFree((char *) color_list);
            XtAddCallback(list_box, XmNdefaultActionCallback,
                color_list_cb, NULL);
            XtManageChild(list_box);
            break;

        case 1:

            // MY 8/5/97
            if ( still_open() )
            {
                warning(drawing_a, "Please close other windows first");
                break;
            }
    }
}

```

```

}

font_list =
    (XmStringTable) XtMalloc(MAXFONTS * sizeof(XmString *));
for(i = 0; i < MAXFONTS; i++)
    font_list[i] =
        XmStringCreateSimple(graphic_list.font_name(i + 1));
list_box =
    XmCreateScrolledList(drawing_a, "Fonts", NULL, 0);
XtVaSetValues(list_box,
    XmNitems, font_list,
    XmNitemCount, MAXFONTS,
    XmNvisibleItemCount, 7,
    NULL);
for(i = 0; i < MAXFONTS; i++)
    XmStringFree(font_list[i]);
XtFree((char *) font_list);
XtAddCallback(list_box, XmNdefaultActionCallback,
    font_list_cb, NULL);
XtManageChild(list_box);
break;

```

case 2:

```

// MY 8/5/97
if ( still_open() )
{
    warning(drawing_a, "Please close other windows first");
    break;
}

if (selected_object_ptr != NULL) {
    selected_object_ptr->unselect();
    selected_object_ptr = NULL;
}
graphic_list.get_del_op_list(del_op_str, del_op_id,
    num_del_ops);

op_list = (XmStringTable)
    XtMalloc((num_del_ops + 1) * sizeof(XmString *));
for(i = 0; i < num_del_ops; i++)
    op_list[i] = XmStringCreateSimple(del_op_str[i]);
op_list[num_del_ops] = XmStringCreateSimple("Cancel");
op_box = XmCreateScrolledList(drawing_a, "Undelete",
    NULL, 0);

XtVaSetValues(op_box,
    XmNitems, op_list,
    XmNitemCount, num_del_ops + 1,
    XmNvisibleItemCount, 7,
    NULL);
for(i = 0; i < num_del_ops + 1; i++)
    XmStringFree(op_list[i]);
XtFree((char *) op_list);
XtAddCallback(op_box, XmNdefaultActionCallback,
    op_list_cb, NULL);
XtManageChild(op_box);
break;

```

```

/*
* MY
*

```

case 3:

```

// MY 8/5/97
if ( still_open() )
{
    warning(drawing_a, "Please close other windows first");
    break;
}

Quest_Script abandon_script =
    {"", "All changes will be lost, are you sure?",
     "Yes", "No", "Cancel", BTN1};

reply = AskUser(app, drawing_a, abandon_script);
if (reply == YES) {
    next_action_ptr->option      = ABANDON;
    next_action_ptr->reinvoke    = true;
    free(next_action_ptr->next_op);
    next_action_ptr->next_op     = graphic_list.current_op_name();
    next_action_ptr->next_op_num = graphic_list.current_op_num();
    return_sde_flag             = true;
}

break;

*/

case 3:
    XFillRectangle(display_ptr, drawing_area_pixmap,
                    erase_context, 0, 0, width, height);
    XFillRectangle(display_ptr, draw_window,
                    erase_context, 0, 0, width, height);
    graphic_list.draw();

    break;

default:
    break;
}
XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);
}

void handle_layout_options(int item_no) {
    int i, x, y, node_count;
    int to_id, from_id, icount;
    GraphObject *temp_ptr, *next_ptr;
    OperatorObject *operators[500];
    OperatorObject *obj_ptr;
    StreamObject *str_ptr;
    SplineObject *spl_ptr, spline;
    graph G;
    node* V = new node[500];

    // BUILD LEDA GRAPH FROM CAPS GRAPH
    node_count = 0;
    temp_ptr = (GraphObject *) graphic_list.cur_graph();
    G.clear();
    while (temp_ptr != NULL)
    {
        if (temp_ptr->is_a() == OPERATOROBJECT)
        {
            obj_ptr = (OperatorObject *) temp_ptr;

```



```

V[node_count] = G.new_node();
operators[node_count] = obj_ptr;
obj_ptr->set_location(x,y);
obj_ptr->set_default_text_location();
graphic_list.move_notify(obj_ptr->is_a(), obj_ptr->id());
cout << ".";
node_count++;
}
else if (temp_ptr->is_a() == STREAMOBJECT)
{
    str_ptr = (StreamObject *) temp_ptr;
    to_id = -9;
    from_id = -9;
    for (icount=0; icount<node_count; icount++)
    {
        if (str_ptr->to() == operators[icount]->id())
            to_id = icount;
        if (str_ptr->from() == operators[icount]->id())
            from_id = icount;
    }
    if ( (to_id >= 0) && (to_id < node_count) &&
        (from_id >= 0) && (from_id < node_count) )
    {
        G.new_edge(V[from_id], V[to_id]);
        cout << "-";
    }
    else cout << "ERROR PARSING GRAPH EDGE in handle_layout";
}
else
{
    cout << "?";
}
next_ptr = temp_ptr->next();
temp_ptr = next_ptr;
}
cout << endl;

// RUN ALGORITHM
switch (item_no) {
    case 0:
        cout << "Got orthoganal layout parameter" << endl;
        cout << "Got " << item_no << " parameter" << endl;
        run_ortho(G, TRUE);
        break;
    case 1:
        cout << "Got straight line 2 layout parameter" << endl;
        cout << "Got " << item_no << " parameter" << endl;
        run_straight_line2(G, TRUE);
        break;
    case 2:
        cout << "Got straight line layout parameter" << endl;
        cout << "Got " << item_no << " parameter" << endl;
        run_straight_line(G, TRUE);
        break;
    case 3:
        cout << "Got tutte layout parameter" << endl;
        cout << "Got " << item_no << " parameter" << endl;
        run_tutte(G, TRUE);
        break;
    case 4:

```

```

        cout << "Got D2 spring embedder layout parameter" << endl;
        cout << "Got " << item_no << " parameter" << endl;
        run_d2_spring_embedding(G, TRUE);
        break;
    case 5:
        cout << "Got spring embedder layout parameter" << endl;
        cout << "Got " << item_no << " parameter" << endl;
        run_spring_embedding(G, TRUE);
        break;
    default:
        cout << "Got undefined parameter" << endl;
        break;
}

// SET CAPS POSITIONS

// DRAW GRAPH
graphic_list.draw();
// graphic_list.draw();
// XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);
// warning(drawing_a, "Not yet implemented.");
}

void handle_tool_options(int item_no) {
    warning(drawing_a, "Not yet implemented.");
}

void set_color(Widget widget, char *color) {
    Display *dpy = XtDisplay(widget);
    Colormap cmap = DefaultColormapOfScreen(XtScreen(widget));
    XColor col, unused;

    if (!XAllocNamedColor(dpy, cmap, color, &col, &unused)) {
        warning(drawing_a, "Can't allocate color");
        return;
    }
    XSetForeground(dpy, std_graphics_context, col.pixel);
}

/*****
 * Menu call-back functions. These functions are called by the window
 * manager when a menu option is selected from a pull-down menu. The
 * item which was selected is passed in client_data.
 *****/

static void file_menu_cb(Widget, XtPointer client_data, XtPointer) {
    int item_no = (int) client_data;

    handle_file_options(item_no);
}

static void psdl_menu_cb(Widget, XtPointer client_data, XtPointer) {
    int item_no = (int) client_data;

    handle_psdl_options(item_no);
}

static void edit_menu_cb(Widget, XtPointer client_data, XtPointer) {

```

```

    int item_no = (int) client_data;

    handle_edit_options(item_no);
}

static void layout_menu_cb(Widget, XtPointer client_data, XtPointer) {
    int item_no = (int) client_data;

    handle_layout_options(item_no);
}

static void tool_menu_cb(Widget, XtPointer client_data, XtPointer) {
    int item_no = (int) client_data;

    handle_tool_options(item_no);
}

static void help_menu_cb(Widget w, XtPointer client_data,
                        XtPointer call_data) {
    int item_no = (int) client_data;

    help_cb(drawing_a, help_menu_files[item_no], call_data);
}

void help_cb(Widget w, XtPointer client_data, XtPointer call_data) {
// Implemented by Doug Lange 8/19/96

    Widget help_dialog, pane, text_w, rc, action_a;
    struct stat statb;

    char ch, *buf;
    int i = 0, n = 0;
    int len = 0;
    Arg      args[10];

    static ActionAreaItem  action_items[] = {
        {"OK", close_dialog, NULL}
    };

    help_dialog = XtVaCreatePopupShell("Help",
                                       xmDialogShellWidgetClass, XtParent(w),
                                       XmNdeleteResponse, XmDESTROY,
                                       NULL);

    pane = XtVaCreateWidget("pane", xmPanedWindowWidgetClass, help_dialog,
                           XmNsashWidth, 1,
                           XmNsashHeight, 1,
                           NULL);

    rc = XtVaCreateWidget("control_area", xmRowColumnWidgetClass, pane,
                          NULL);

    stat((char*)client_data, &statb);
    ifstream from((char *)client_data);
    len = statb.st_size;
    buf = new char[len+1]; // Add a space for NULL
    i = 0;
    // while (from.get(ch) && (i < HELPSIZ -1)) {
    while (from.get(ch) && (i < len)) {
        buf[i] = ch;

```

```

        i++;
    }
    buf[i] = (char)NULL;

    XtSetArg(args[n], XmNscrollVertical,      true); n++;
    XtSetArg(args[n], XmNscrollHorizontal,    false); n++;
    XtSetArg(args[n], XmNeditMode,            XmMULTI_LINE_EDIT); n++;
    XtSetArg(args[n], XmNeditable,            false); n++;
    XtSetArg(args[n], XmNcursorPositionVisible, false); n++;
    XtSetArg(args[n], XmNwordWrap,            true); n++;
    XtSetArg(args[n], XmNvalue,                buf); n++;
    XtSetArg(args[n], XmNrows,                20); n++;
    XtSetArg(args[n], XmNwidth,               525); n++;
    text_w = XmCreateScrolledText(rc, "help_text", args, n);

    delete buf;

    XtManageChild(text_w);
    XtManageChild(rc);

    action_items[0].data = (XtPointer)help_dialog;
    action_a = CreateActionArea(pane, action_items,
    XtNumber(action_items));

    XtManageChild(pane);
    XtPopup(help_dialog, XtGrabNone);
}

void build_menu_bar(Widget &main_w, Widget &menubar) {
    // 8/4/96 KBM Updated for label changes in Req 4
    // Also changed callback names to reflect new labels.

    // ?? Need to look at short-cut keys....not implemented correctly

    XmString
        file_menu, save_opt, restore_opt, print_opt, exit_opt,
        psdl_menu, syntax_check_opt, goto_root_opt, goto_parent_opt,
        decompose_opt,
        edit_menu, color_opt, font_opt, undelete_opt, abandon_opt,
refresh_opt,
// tool_menu, reuse_lib_opt,
        layout_menu, ortho_opt, str_line2_opt, str_line_opt, tutte_opt,
d2_se_opt,
        spring_opt,
        help_menu, psdl_grammar_opt, operator_opt, stream_opt,
exception_opt,
        timer_opt;

    Widget widget;

    file_menu      = XmStringCreateSimple("File");
    save_opt       = XmStringCreateSimple("Save");
    restore_opt    = XmStringCreateSimple("Restore from Save");
    print_opt      = XmStringCreateSimple("Print");
    exit_opt       = XmStringCreateSimple("Exit");

    psdl_menu      = XmStringCreateSimple("PSDL");
    syntax_check_opt = XmStringCreateSimple("Syntax Check");
    goto_root_opt  = XmStringCreateSimple("Go to Root");

```

```

goto_parent_opt = XmStringCreateSimple("Go to Parent");
decompose_opt   = XmStringCreateSimple("Decompose");

edit_menu       = XmStringCreateSimple("Edit");
color_opt       = XmStringCreateSimple("Color");
font_opt        = XmStringCreateSimple("Font");
undelete_opt    = XmStringCreateSimple("Undelete Operator");
abandon_opt     = XmStringCreateSimple("Abandon Changes");
refresh_opt     = XmStringCreateSimple("Refresh Display");
/*
tool_menu       = XmStringCreateSimple("Tools");
reuse_lib_opt   = XmStringCreateSimple("Reuse Library");
*/
printf("Just before layout menu.\n");

layout_menu     = XmStringCreateSimple("Layout");
ortho_opt       = XmStringCreateSimple("Orthogonal Layout");
str_line2_opt   = XmStringCreateSimple("Straight Line 2");
str_line_opt    = XmStringCreateSimple("Straight Line");
tutte_opt       = XmStringCreateSimple("Tutte Layout");
d2_se_opt       = XmStringCreateSimple("D2 Spring Embedding");
spring_opt      = XmStringCreateSimple("Spring Embedding");

help_menu       = XmStringCreateSimple("Help");
psdl_grammar_opt = XmStringCreateSimple("PSDL Grammar");
operator_opt     = XmStringCreateSimple("Operators");
stream_opt       = XmStringCreateSimple("Streams");
exception_opt    = XmStringCreateSimple("Exceptions");
timer_opt        = XmStringCreateSimple("Timers");

menubar = XmVaCreateSimpleMenuBar(main_w, "menubar",
    XmVaCASCADEBUTTON, file_menu, NULL,
    XmVaCASCADEBUTTON, psdl_menu, NULL,
    XmVaCASCADEBUTTON, edit_menu, NULL,
//    XmVaCASCADEBUTTON, tool_menu, NULL,
    XmVaCASCADEBUTTON, layout_menu, NULL,
    XmVaCASCADEBUTTON, help_menu, NULL, NULL);

if (widget = XtNameToWidget(menubar, "button_4")) // Assign
to help
    XtVaSetValues(menubar, XmNmenuHelpWidget, widget, NULL);

XmVaCreateSimplePulldownMenu(menubar, "file_menu", 0, file_menu_cb,
    XmVaPUSHBUTTON, save_opt,      NULL, NULL, NULL,
    XmVaPUSHBUTTON, restore_opt,    NULL, NULL, NULL,
    XmVaPUSHBUTTON, print_opt,      NULL, NULL, NULL,
/*
* 7/22/97 MY
*/
    XmVaPUSHBUTTON, abandon_opt,    NULL, NULL, NULL,

    XmVaPUSHBUTTON, exit_opt,       NULL, NULL, NULL,
    NULL);

XmVaCreateSimplePulldownMenu(menubar, "psdl_menu", 1, psdl_menu_cb,
/*
* 7/22/97 MY
*
* XmVaPUSHBUTTON, syntax_check_opt, NULL, NULL, NULL,
*/
    XmVaPUSHBUTTON, goto_root_opt,  'R', NULL, NULL,

```

```

        XmVaPUSHBUTTON, goto_parent_opt,    'P', NULL, NULL,
        XmVaPUSHBUTTON, decompose_opt,     'D', NULL, NULL,
        NULL);

XmVaCreateSimplePulldownMenu(menubar, "edit_menu", 2, edit_menu_cb,
        XmVaPUSHBUTTON, color_opt,         NULL, NULL, NULL,
        XmVaPUSHBUTTON, font_opt,          NULL, NULL, NULL,
        XmVaPUSHBUTTON, undelete_opt,      NULL, NULL, NULL,
/*
* 7/22/97 MY
*
* XmVaPUSHBUTTON, abandon_opt,             NULL, NULL, NULL,
*/
        XmVaPUSHBUTTON, refresh_opt,       'f', NULL, NULL,
        NULL);

/*
XmVaCreateSimplePulldownMenu(menubar, "tool_menu", 3, tool_menu_cb,
        XmVaPUSHBUTTON, reuse_lib_opt,     'U', NULL, NULL,
        NULL);
*/

XmVaCreateSimplePulldownMenu(menubar, "layout_menu", 3,
layout_menu_cb,
        XmVaPUSHBUTTON, ortho_opt,         NULL, NULL, NULL,
        XmVaPUSHBUTTON, str_line2_opt,     NULL, NULL, NULL,
        XmVaPUSHBUTTON, str_line_opt,      NULL, NULL, NULL,
        XmVaPUSHBUTTON, tutte_opt,         NULL, NULL, NULL,
        XmVaPUSHBUTTON, d2_se_opt,         NULL, NULL, NULL,
        XmVaPUSHBUTTON, spring_opt,        NULL, NULL, NULL,
        NULL);

XmVaCreateSimplePulldownMenu(menubar, "help_menu", 4, help_menu_cb,
        XmVaPUSHBUTTON, psdl_grammar_opt,  NULL, NULL, NULL,
        XmVaPUSHBUTTON, operator_opt,      NULL, NULL, NULL,
        XmVaPUSHBUTTON, stream_opt,        NULL, NULL, NULL,
        XmVaPUSHBUTTON, exception_opt,     NULL, NULL, NULL,
        XmVaPUSHBUTTON, timer_opt,         NULL, NULL, NULL,
        NULL);

XmStringFree(file_menu);
XmStringFree(save_opt);
XmStringFree(restore_opt);
XmStringFree(print_opt);
XmStringFree(exit_opt);
XmStringFree(psdl_menu);
XmStringFree(syntax_check_opt);
XmStringFree(goto_root_opt);
XmStringFree(goto_parent_opt);
XmStringFree(decompose_opt);
XmStringFree(edit_menu);
XmStringFree(color_opt);
XmStringFree(font_opt);
XmStringFree(undelete_opt);
XmStringFree(abandon_opt);
XmStringFree(refresh_opt);
//XmStringFree(tool_menu);
// XmStringFree(reuse_lib_opt);
XmStringFree(layout_menu);
XmStringFree(ortho_opt);
XmStringFree(str_line2_opt);

```

```

    XmStringFree(str_line_opt);
    XmStringFree(tutte_opt);
    XmStringFree(d2_se_opt);
    XmStringFree(spring_opt);
    XmStringFree(help_menu);
    XmStringFree(psdl_grammar_opt);
    XmStringFree(operator_opt);
    XmStringFree(stream_opt);
    XmStringFree(exception_opt);
    XmStringFree(timer_opt);
}

//  Creates the push buttons used to select the tools.

void make_buttons(Widget &rowcol,
                  Widget &op_button,           // tools
                  Widget &term_button,
                  Widget &stream_button,
                  Widget &select_button,
                  Widget &types_button,        // types
                  Widget &spec_button,         // current op spec
                  Widget &timers_button,       // current op impl
                  Widget &informal_button,     // current op impl

                  Pixmap &op_button_pixmap,
                  Pixmap &term_button_pixmap,
                  Pixmap &stream_button_pixmap,
                  Pixmap &select_button_pixmap,
                  Pixmap &types_button_pixmap,
                  Pixmap &spec_button_pixmap,
                  Pixmap &timers_button_pixmap,
                  Pixmap &informal_button_pixmap,
                  Display *display_ptr,
                  Screen *screen_ptr) {

    static Widget op_btn_bb, term_btn_bb, stream_btn_bb, select_btn_bb,
                  types_btn_bb, spec_btn_bb, timers_btn_bb,
informal_btn_bb;

    Window root_window = RootWindowOfScreen(screen_ptr);
    unsigned int screen_depth = DefaultDepthOfScreen(screen_ptr);

    op_button_pixmap      = XCreatePixmap(display_ptr, root_window,
                                          BUTTONWIDTH-4, BUTTONWIDTH-4, screen_depth);
    term_button_pixmap    = XCreatePixmap(display_ptr, root_window,
                                          BUTTONWIDTH-4, BUTTONWIDTH-4, screen_depth);
    stream_button_pixmap  = XCreatePixmap(display_ptr, root_window,
                                          BUTTONWIDTH-4, BUTTONWIDTH-4, screen_depth);
    select_button_pixmap  = XCreatePixmap(display_ptr, root_window,
                                          BUTTONWIDTH-4, BUTTONWIDTH-4, screen_depth);
    types_button_pixmap   = XCreatePixmap(display_ptr, root_window,
                                          BUTTONWIDTH-4, BUTTONWIDTH-4, screen_depth);
    spec_button_pixmap    = XCreatePixmap(display_ptr, root_window,
                                          BUTTONWIDTH-4, BUTTONWIDTH-4, screen_depth);
    timers_button_pixmap  = XCreatePixmap(display_ptr, root_window,
                                          BUTTONWIDTH-4, BUTTONWIDTH-4, screen_depth);
    informal_button_pixmap = XCreatePixmap(display_ptr, root_window,
                                          BUTTONWIDTH-4, BUTTONWIDTH-4, screen_depth);

    XFillRectangle(display_ptr, (Drawable) op_button_pixmap,

```

```

        erase_context, 0, 0, BUTTONWIDTH-4, BUTTONWIDTH-4);
XFillRectangle(display_ptr, (Drawable) term_button_pixmap,
        erase_context, 0, 0, BUTTONWIDTH-4, BUTTONWIDTH-4);
XFillRectangle(display_ptr, (Drawable) stream_button_pixmap,
        erase_context, 0, 0, BUTTONWIDTH-4, BUTTONWIDTH-4);
XFillRectangle(display_ptr, (Drawable) select_button_pixmap,
        erase_context, 0, 0, BUTTONWIDTH-4, BUTTONWIDTH-4);
XFillRectangle(display_ptr, (Drawable) types_button_pixmap,
        erase_context, 0, 0, BUTTONWIDTH-4, BUTTONWIDTH-4);
XFillRectangle(display_ptr, (Drawable) spec_button_pixmap,
        erase_context, 0, 0, BUTTONWIDTH-4, BUTTONWIDTH-4);
XFillRectangle(display_ptr, (Drawable) timers_button_pixmap,
        erase_context, 0, 0, BUTTONWIDTH-4, BUTTONWIDTH-4);
XFillRectangle(display_ptr, (Drawable) informal_button_pixmap,
        erase_context, 0, 0, BUTTONWIDTH-4, BUTTONWIDTH-4);

XSetLineAttributes(display_ptr, std_graphics_context, 2,
        LineSolid, CapButt, JoinMiter);

XDrawArc(display_ptr, (Drawable) op_button_pixmap,
        std_graphics_context,
        10, 15, BUTTONWIDTH-(3*10), BUTTONWIDTH-(3*10),
        CIRCLE_BEGIN, FULL_CIRCLE);
XDrawRectangle(display_ptr, (Drawable) term_button_pixmap,
        std_graphics_context,
        10, 15, BUTTONWIDTH-(3*10), BUTTONWIDTH-(3*10));
XDrawLine(display_ptr, (Drawable) stream_button_pixmap,
        std_graphics_context,
        10, 15, BUTTONWIDTH-(2*10), BUTTONWIDTH-(2*10));
XDrawString(display_ptr, (Drawable) select_button_pixmap,
        std_graphics_context, 10, (BUTTONWIDTH/2)+5, "Select", 6);
XDrawString(display_ptr, (Drawable) types_button_pixmap,
        std_graphics_context, 10, (BUTTONWIDTH/2)+5, "Types ", 6);
XDrawString(display_ptr, (Drawable) spec_button_pixmap,
        std_graphics_context, 10, (BUTTONWIDTH/2)+5, " Spec ", 6);
XDrawString(display_ptr, (Drawable) timers_button_pixmap,
        std_graphics_context, 10, (BUTTONWIDTH/2)+5, "Timers", 6);
XDrawString(display_ptr, (Drawable) informal_button_pixmap,
        std_graphics_context, 10, (BUTTONWIDTH/2)-8, "Graph ", 6);
XDrawString(display_ptr, (Drawable) informal_button_pixmap,
        std_graphics_context, 5, (BUTTONWIDTH/2)+5, "Informal",
8);
XDrawString(display_ptr, (Drawable) informal_button_pixmap,
        std_graphics_context, 10, (BUTTONWIDTH/2)+18, "Desc ",
6);

XmString button_label;

button_label = XmStringCreateSimple("Operator");

op_button = XtVaCreateManagedWidget("op_button",
        xmDrawnButtonWidgetClass,
        rowcol,
        XmNrecomputeSize, false,
        XmNpushButtonEnabled, false,
        XmNshadowType, XmSHADOW_OUT,
        XmNwidth, BUTTONWIDTH,
        XmNheight, BUTTONWIDTH,
        XmNlabelType, XmSTRING,
        XmNlabelString, button_label,
        //XmNlabelType, XmPIXMAP,

```



```

        //XmNlabelPixmap, op_button_pixmap,
        NULL);

XmStringFree(button_label);
button_label = XmStringCreateSimple(" Term");

term_button = XtVaCreateManagedWidget("term_button",
        xmDrawnButtonWidgetClass,
        rowcol,
        XmNrecomputeSize, false,
        XmNpushButtonEnabled, false,
        XmNshadowType, XmSHADOW_OUT,
        XmNwidth, BUTTONWIDTH,
        XmNheight, BUTTONWIDTH,
        XmNlabelType, XmSTRING,
        XmNlabelString, button_label,
        //XmNlabelType, XmPIXMAP,
        //XmNlabelPixmap, op_button_pixmap,
        NULL);

XmStringFree(button_label);
button_label = XmStringCreateSimple(" Stream");

stream_button = XtVaCreateManagedWidget("stream_button",
        xmDrawnButtonWidgetClass,
        rowcol,
        XmNrecomputeSize, false,
        XmNpushButtonEnabled, false,
        XmNshadowType, XmSHADOW_OUT,
        XmNwidth, BUTTONWIDTH,
        XmNheight, BUTTONWIDTH,
        XmNlabelType, XmSTRING,
        XmNlabelString, button_label,
        //XmNlabelType, XmPIXMAP,
        //XmNlabelPixmap, op_button_pixmap,
        NULL);

XmStringFree(button_label);
button_label = XmStringCreateSimple(" Select");

select_button = XtVaCreateManagedWidget("select_button",
        xmDrawnButtonWidgetClass,
        rowcol,
        XmNrecomputeSize, false,
        XmNpushButtonEnabled, false,
        XmNshadowType, XmSHADOW_OUT,
        XmNwidth, BUTTONWIDTH,
        XmNheight, BUTTONWIDTH,
        XmNlabelType, XmSTRING,
        XmNlabelString, button_label,
        //XmNlabelType, XmPIXMAP,
        //XmNlabelPixmap, op_button_pixmap,
        NULL);

XmStringFree(button_label);

button_divider = XtVaCreateManagedWidget("separator",
        xmSeparatorWidgetClass, rowcol,
//      XmNy, ROW14 - 8,
//      XmNwidth, WIN_WIDTH,
        NULL);

```

```

button_label = XmStringCreateSimple("  Types");

types_button = XtVaCreateManagedWidget("types_button",
    xmDrawnButtonWidgetClass,
    rowcol,
    XmNrecomputeSize, false,
    XmNpushButtonEnabled, false,
    XmNshadowType, XmSHADOW_OUT,
    XmNwidth, BUTTONWIDTH,
    XmNheight, BUTTONWIDTH,
    XmNlabelType, XmSTRING,
    XmNlabelString, button_label,
    //XmNlabelType, XmPIXMAP,
    //XmNlabelPixmap, op_button_pixmap,
    NULL);

XmStringFree(button_label);
button_label = XmStringCreateLtoR(" Parent\n Spec",
    XmSTRING_DEFAULT_CHARSET);

spec_button = XtVaCreateManagedWidget("spec_button",
    xmDrawnButtonWidgetClass,
    rowcol,
    XmNrecomputeSize, false,
    XmNpushButtonEnabled, false,
    XmNshadowType, XmSHADOW_OUT,
    XmNwidth, BUTTONWIDTH,
    XmNheight, BUTTONWIDTH,
    XmNlabelType, XmSTRING,
    XmNlabelString, button_label,
    //XmNlabelType, XmPIXMAP,
    //XmNlabelPixmap, op_button_pixmap,
    NULL);

XmStringFree(button_label);
button_label = XmStringCreateSimple(" Timers");

timers_button = XtVaCreateManagedWidget("timers_button",
    xmDrawnButtonWidgetClass,
    rowcol,
    XmNrecomputeSize, false,
    XmNpushButtonEnabled, false,
    XmNshadowType, XmSHADOW_OUT,
    XmNwidth, BUTTONWIDTH,
    XmNheight, BUTTONWIDTH,
    XmNlabelType, XmSTRING,
    XmNlabelString, button_label,
    //XmNlabelType, XmPIXMAP,
    //XmNlabelPixmap, op_button_pixmap,
    NULL);

XmStringFree(button_label);
button_label = XmStringCreateLtoR(" Graph\n Desc",
    XmSTRING_DEFAULT_CHARSET);

informal_button = XtVaCreateManagedWidget("informal_button",
    xmDrawnButtonWidgetClass,
    rowcol,
    XmNrecomputeSize, false,

```

```

        XmNpushButtonEnabled, false,
        XmNshadowType, XmSHADOW_OUT,
        XmNwidth, BUTTONWIDTH,
        XmNheight, BUTTONHEIGHT,
        XmNlabelType, XmSTRING,
        XmNlabelString, button_label,
        //XmNlabelType, XmPIXMAP,
        //XmNlabelPixmap, op_button_pixmap,
        NULL);
XmStringFree(button_label);

XSetLineAttributes(display_ptr, std_graphics_context, 1,
                    LineSolid, CapButt, JoinMiter);
}

// Redraws the drawing canvas.

void redraw(Widget, XtPointer,
            XtPointer cbs) {
    XmDrawingAreaCallbackStruct *temp_ptr;

    temp_ptr = (XmDrawingAreaCallbackStruct *) cbs;
    XCopyArea(temp_ptr->event->xexpose.display,
              drawing_area_pixmap, temp_ptr->window,
              std_graphics_context, 0, 0, width, height, 0, 0);
}

// Draws a square black box on the canvas to aid in
// graphic manipulation of objects.

void draw_handle(GC graphics_context, int x, int y) {
    x -= HANDLESIZE / 2;
    y -= HANDLESIZE / 2;
    if (x < 0)
        x = 0;
    if (y < 0)
        y = 0;

    // When the display function is set to GXxor, the pixel being
    // written is exclusive-or'ed with the target pixel to
    // determine color. This means that writing the same pixel with
    // the same color twice restores the original color, simplifying
    // the process of erasing handles.

    XSetFunction(display_ptr, graphics_context, GXxor);
    XFillRectangle(display_ptr, draw_window, graphics_context,
                   x, y, HANDLESIZE, HANDLESIZE);
    XFillRectangle(display_ptr, drawing_area_pixmap,
                   graphics_context,
                   x, y, HANDLESIZE, HANDLESIZE);
    XSetFunction(display_ptr, graphics_context, GXcopy);
}

// This function erases the temporary guidelines used when
// streams are drawn.
// Dotted lines are erased first, then handles. Since each
// handle is overwritten with the following dotted line, an
// erased handle makes an erased blotch in the beginning of the

```

```

// next segment. When the next segment is written in xor mode,
// it makes a black mark where the erased handle overwrote the
// beginning of its segment.

void erase_guides(OP_ID from_stream_id, SplineObject *temp_spline_ptr) {
    OperatorObject *temp_operator_ptr;
    XYPAIR line_start, line_end;

    temp_spline_ptr->reset_iter();
    if (from_stream_id != UNDEFINED_OPNUM) {
        temp_operator_ptr = (OperatorObject *)
            graphic_list.target_object(OPERATOROBJECT, from_stream_id);
        line_start = temp_operator_ptr->center();
    }
    else
        line_start = temp_spline_ptr->next_pair();
    line_end = temp_spline_ptr->next_pair();
    while(line_end.x != -1) {
        XDrawLine(display_ptr, draw_window, dotted_context,
            line_start.x, line_start.y, line_end.x,
            line_end.y);
        XDrawLine(display_ptr, drawing_area_pixmap, dotted_context,
            line_start.x, line_start.y, line_end.x,
            line_end.y);
        line_start = line_end;
        line_end = temp_spline_ptr->next_pair();
    }
    temp_spline_ptr->reset_iter();
    line_end = temp_spline_ptr->next_pair();
    while(line_end.x != -1) {
        draw_handle(std_graphics_context, line_end.x, line_end.y);
        line_end = temp_spline_ptr->next_pair();
    }
}

```

```

// This function is called when a stream is being drawn
// and the mouse is clicked on either a clear spot on the
// drawing canvas, or on top of another stream. If a double-
// click is registered, the user wants to terminate an external
// stream.

```

```

void handle_null_point(OP_ID from_stream_id, int &last_point_x,
    int &last_point_y,
    int &x_state, int &y_state,
    XEvent in_event,
    SplineObject *temp_spline_ptr, BOOLEAN &done,
    GraphObject *&temp_object_ptr,
    StreamObject *&temp_stream_ptr) {

```

```

// Checks for two clicks in the same spot.

```

```

if ((from_stream_id != UNDEFINED_OPNUM) &&
    ((last_point_x - (HANDLESIZE / 2) - HITFUDGE)
    < in_event.xbutton.x) &&
    ((last_point_x + (HANDLESIZE / 2) + HITFUDGE)
    > in_event.xbutton.x) &&
    ((last_point_y - (HANDLESIZE / 2) - HITFUDGE)
    < in_event.xbutton.y) &&
    ((last_point_y + (HANDLESIZE / 2) + HITFUDGE)
    > in_event.xbutton.y)) {

```

```

erase_guides(from_stream_id, temp_spline_ptr);
OP_ID new_id = graphic_list.request_id(STREAMOBJECT);

// MY 8/4/97
sprintf(default_name, "noname_%d", get_unique_id());
//MY: "" -> default_name
temp_stream_ptr = new StreamObject(default_name, new_id,
                                   from_stream_id,
                                   0, UNDEFINED_TIME, MS,
                                   temp_spline_ptr, // @6
                                   true, false);
temp_stream_ptr->set_object_ptrs(&graphic_list);
graphic_list.add(temp_stream_ptr);
save_state(SAVE_REQUIRED);
temp_stream_ptr->draw(SOLID);
temp_stream_ptr = NULL;
temp_object_ptr = NULL;
done = true;
temp_spline_ptr->clear();
}
else {
    x_state = in_event.xbutton.x;
    y_state = in_event.xbutton.y;
    temp_spline_ptr->add(x_state, y_state);
    XDrawLine(display_ptr, draw_window, dotted_context,
              last_point_x, last_point_y, x_state, y_state);
    XDrawLine(display_ptr, drawing_area_pixmap, dotted_context,
              last_point_x, last_point_y, x_state, y_state);
    draw_handle(std_graphics_context, x_state, y_state);
#ifdef GE_DEBUG
    //      cout << "ge: " << x_state << " " << y_state << " " <<
    //      HANDLESIZE << endl;
#endif
    last_point_x = x_state;
    last_point_y = y_state;
}
}

//  Once the user selects the Stream Tool and begins to draw,
//  the draw_stream() function handles all events to speed up
//  performance.

void draw_stream(int initial_x, int initial_y) {
    GraphObject *temp_object_ptr;
    OperatorObject *conv_ptr;
    XYPAIR temp_pair;
    char buffer[INPUT_LINE_SIZE]; // added for req #6.4 dha
    int count = 0; // added for req #6.4 dha
    int bufsize = INPUT_LINE_SIZE; // added for req #6.4 dha
    OP_ID from_stream_id;
    int x_state, y_state, last_point_x, last_point_y;
    unsigned long stream_event_mask =
        (ButtonPressMask | PointerMotionMask |
         KeyPressMask);
    unsigned long normal_mask = (ButtonPressMask | PointerMotionMask |
                                KeyPressMask |
                                ButtonMotionMask | ExposureMask |
                                ButtonReleaseMask);

    XEvent in_event;
    StreamObject *temp_stream_ptr;

```

```

SplineObject *temp_spline_ptr;
BOOLEAN done = false;
KeySym keysym; // added for req #6.4 dha

temp_spline_ptr = new SplineObject;

temp_object_ptr = graphic_list.hit(initial_x, initial_y);
if (temp_object_ptr == NULL) { // External stream
    from_stream_id = UNDEFINED_OPNUM;
    temp_spline_ptr->add(initial_x, initial_y);
    x_state = initial_x;
    y_state = initial_y;
    draw_handle(std_graphics_context, x_state, y_state);
}
else {
    if (temp_object_ptr->is_a() != OPERATOROBJECT) {
        // External Stream
        from_stream_id = UNDEFINED_OPNUM;
        temp_spline_ptr->add(initial_x, initial_y);
        x_state = initial_x;
        y_state = initial_y;
        draw_handle(std_graphics_context, x_state, y_state);
    }
    else {
        conv_ptr = (OperatorObject *) temp_object_ptr;
        from_stream_id = conv_ptr->id();
        temp_object_ptr = NULL;
        temp_pair = conv_ptr->center();
        x_state = temp_pair.x;
        y_state = temp_pair.y;
    }
}
last_point_x = x_state;
last_point_y = y_state;
XSelectInput(display_ptr, draw_window,
              stream_event_mask);
while(done == false) { // monitors the event loop
    XNextEvent(display_ptr, &in_event);
    if (in_event.xbutton.window == draw_window) {
        switch(in_event.type) {
            case MotionNotify:

#ifdef GE_DEBUG
                // cout << "Motion" << endl;
#endif

                XDrawLine(display_ptr, draw_window, dotted_context,
                           last_point_x, last_point_y, x_state, y_state);
                XDrawLine(display_ptr, drawing_area_pixmap,
                           dotted_context, last_point_x, last_point_y,
                           x_state, y_state);
                x_state = in_event.xbutton.x;
                y_state = in_event.xbutton.y;
                XDrawLine(display_ptr, draw_window, dotted_context,
                           last_point_x, last_point_y, x_state, y_state);
                XDrawLine(display_ptr, drawing_area_pixmap,
                           dotted_context, last_point_x, last_point_y,
                           x_state, y_state);
                break;
            case ButtonPress:
            case KeyPress:

```

```

#ifdef GE_DEBUG
    if (in_event.type == ButtonPress) {
//        cout << "buttonpress" << endl;
    }
    else {
//        cout << "keypress" << endl;
    }
#endif

XDrawLine(display_ptr, draw_window, dotted_context,
           last_point_x, last_point_y, x_state, y_state);
XDrawLine(display_ptr, drawing_area_pixmap,
           dotted_context, last_point_x, last_point_y,
           x_state, y_state);
temp_object_ptr = graphic_list.hit(in_event.xbutton.x,
                                   in_event.xbutton.y);
if (temp_object_ptr == NULL) {
    handle_null_point(from_stream_id, last_point_x,
                     last_point_y, x_state, y_state,
                     in_event, temp_spline_ptr, done,
                     temp_object_ptr, temp_stream_ptr);
}
else
    if (temp_object_ptr->is_a() == OPERATOROBJECT) {
        erase_guides(from_stream_id, temp_spline_ptr);
        OP_ID new_id = graphic_list.request_id(STREAMOBJECT);
// MY 8/4/97
        sprintf(default_name, "noname_%d", get_unique_id());
        temp_stream_ptr =
            //MY: "" -> default_name
            new StreamObject(default_name, new_id,
from_stream_id,
                               temp_object_ptr->id(),
                               UNDEFINED_TIME, MS, // @6
                               temp_spline_ptr, true, false);
        temp_stream_ptr->set_object_ptrs(&graphic_list);
        save_state(SAVE_REQUIRED);
        graphic_list.add(temp_stream_ptr);
        temp_stream_ptr->draw(SOLID);
        temp_stream_ptr = NULL;
        temp_object_ptr = NULL;
        done = true;
        temp_spline_ptr->clear();
    }
    else
        if (temp_object_ptr->is_a() == STREAMOBJECT) {
            handle_null_point(from_stream_id, last_point_x,
                             last_point_y, x_state, y_state,
                             in_event, temp_spline_ptr, done,
                             temp_object_ptr, temp_stream_ptr);
        }
}

if (in_event.type == KeyPress) {
    count = XLookupString(&in_event.xkey, buffer,
                          bufsize, &keysym, NULL);
    buffer[count] = NULL; /* add NULL terminator */

    if (keysym == XK_Escape) {
        //temp_stream_ptr->erase();
        //OP_ID deleted_op_id = temp_stream_ptr->id();
        //graphic_list.delete_notify(temp_stream_ptr->

```

```

        //                                is_a(), deleted_op_id);
        //temp_stream_ptr->set_deleted();
        //temp_stream_ptr = NULL;
        graphic_list.draw();

        done = true;
    }
    else {
        XBell(display_ptr, 100);
    }
}
break;

default:
    ;
    break;
} //switch
} //if right window
} //while done == false
done = false;
XSelectInput(display_ptr, draw_window, normal_mask);
}

```

// Draws the outline of the text being moved.

```

void draw_text_shadow(int x, int y, int width, int height) {
    XDrawRectangle(display_ptr, drawing_area_pixmap,
                    dotted_context, x - width / 2, y - height / 2,
                    width, height);
    XDrawRectangle(display_ptr, draw_window, dotted_context,
                    x - width / 2, y - height / 2, width, height);
}

```

// The main draw routine. This function is called by the
// window manager every time the mouse is moved, a mouse button
// pressed, or a key pressed inside the draw window. It is
// called with a string token that indicates why it was called,
// and processes the event accordingly.

```

void draw(Widget, XEvent *event, String *args, Cardinal *) { // void
draw
    static char string[INPUT_LINE_SIZE]; // added for req #6.1.1 dha
    static OperatorObject *temp_operator_ptr = NULL;
    static StreamObject *temp_stream_ptr = NULL;
    static BOOLEAN first_draw = true, handle_selected = false,
                    text_selected = false, drawing_changed = false;
    static int x_state, y_state, shadow_height, shadow_width;
    static OP_ID from_stream_id;
    GraphObject *temp_object_ptr = NULL;
    static GraphObject *ibar_object_ptr = NULL;
    char *warningMSG;
    char buffer[INPUT_LINE_SIZE]; // added for req #6.1.1 dha
    int count = 0; // added for req #6.1.1 dha
    int length = 0; // added for req #6.1.1 dha
    int bufsize = INPUT_LINE_SIZE; // added for req #6.1.1 dha
    int x = event->xbutton.x;
    int y = event->xbutton.y;
    char *labelName;

```



```

OperatorObject *conv_op_ptr;
StreamObject *conv_st_ptr;
KeySym keysym; // added for req #6.1.1 dha
BOOLEAN state_change, type_match;
BOOLEAN type_operator;

if (strcmp(args[0], "down") == 0) { // Button pressed
    clear_status();
    XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);
    x_state = x;
    y_state = y;
    if (tool_state == SELECT_TOOL) {
        if (selected_object_ptr != NULL) {
            if (selected_object_ptr->hit_handle(x, y)) {
                handle_selected = true;
                object_def = selected_object_ptr->is_a();
                if (object_def == OPERATOROBJECT) {
                    op_being_updated = (OperatorObject *)selected_object_ptr;
                    delete temp_operator_ptr;
                    conv_op_ptr = (OperatorObject *) selected_object_ptr;
                    temp_operator_ptr =
                        new OperatorObject("", UNDEFINED_OPNUM, UNDEFINED_OPNUM,
                            UNDEFINED_TIME, MS, conv_op_ptr->x(), // @6
                                conv_op_ptr->y(),
                                conv_op_ptr->radius(),
                                default_color, false,
                                conv_op_ptr->is_composite(),
                                conv_op_ptr->is_terminator());
                    temp_operator_ptr->set_handle_selected(
                        conv_op_ptr->handle_selected());
                } else if (object_def == STREAMOBJECT) {
                    st_being_updated = (StreamObject *)selected_object_ptr;
                }
            } // selected_object_ptr->hit_handle()
            else { // Unselects previously selected object
                handle_selected = false;
                selected_object_ptr->unselect();
                selected_object_ptr = NULL;
                delete temp_operator_ptr;
                temp_operator_ptr = NULL;
            }
        } // selected_object_ptr != NULL

        if (handle_selected == false) {
            temp_object_ptr = graphic_list.hit(x, y);
            if (temp_object_ptr != NULL) {
                temp_object_ptr->select();
                selected_object_ptr = temp_object_ptr;
                text_selected = selected_object_ptr->text_selected();
                object_def = temp_object_ptr->is_a();
                if (object_def == OPERATOROBJECT) {
                    op_being_updated = (OperatorObject *) temp_object_ptr;

                    // Makes temporary operator to move around
                    delete temp_operator_ptr;
                    conv_op_ptr = (OperatorObject *) temp_object_ptr;
                    temp_operator_ptr =
                        new OperatorObject("", UNDEFINED_OPNUM, UNDEFINED_OPNUM,
                            UNDEFINED_TIME, MS, conv_op_ptr->x(), // @6
                                conv_op_ptr->y(),
                                conv_op_ptr->radius(),

```

```

        default_color, false,
        conv_op_ptr->is_composite(),
        conv_op_ptr->is_terminator());
    } else if (object_def == STREAMOBJECT) {
        st_being_updated = (StreamObject *) temp_object_ptr;
    }
} // temp_object_ptr != NULL
else { // No object selected
    temp_object_ptr = NULL;
    selected_object_ptr = NULL;
    delete temp_operator_ptr;
    temp_operator_ptr = NULL;
}
} // handle_selected == false
} /// tool_state == SELECT_TOOL
else { // button down, operator tool selected?
    // if (((tool_state == OPERATOR_TOOL) ||
    //      (tool_state == TERMINATOR_TOOL)) &&
    //      (ibar_mode != true)) || // added
8/22/96 dha,
    // (object_def != OPERATOROBJECT)) { // req. 6.2 &
6.3
    if ((tool_state == OPERATOR_TOOL) ||
        (tool_state == TERMINATOR_TOOL)) {
        OP_ID new_id = graphic_list.request_id(OPERATOROBJECT);
        OP_ID new_op = graphic_list.request_id(OPERATOROBJECT);
        if (tool_state == OPERATOR_TOOL) {
            // MY 8/4/97
            sprintf(default_name, "noname_%d", get_unique_id());
            temp_operator_ptr =
            // BROCKETT 1/22/93 default x and y values changed from 0 to
100
            new OperatorObject(default_name, new_id, new_op, //MY: "" ->
default_name
                                UNDEFINED_TIME, MS, 100, 100, 30, // @6
                                default_color, true, false,
                                false);
            temp_operator_ptr->set_location(x, y);
        } // tool_sate == OPERATOR_TOOL
        else
            if (tool_state == TERMINATOR_TOOL) {
                // MY 8/4/97
                sprintf(default_name, "noname_%d", get_unique_id());
                temp_operator_ptr =
                // BROCKETT 1/22/93 default x and y values changed from 0 to
100
                new OperatorObject(default_name, new_id, new_op, //MY: ""
-> default_p
                                UNDEFINED_TIME, MS, 100, 100, 30, // @6
                                default_color, true, false,
                                true);
                temp_operator_ptr->set_location(x, y);
            } // tool_state == TERMINATOR_TOOL
            graphic_list.add((GraphObject *) temp_operator_ptr);
            save_state(SAVE_REQUIRED);
            temp_operator_ptr->draw(SOLID);
            temp_operator_ptr = NULL;
        } // tool_state == OPERATOR_TOOL || TERMINATOR_TOOL && ibar_mode
11
    else // button down, stream tool selected?
        if (tool_state == STREAM_TOOL) {

```

```

        draw_stream(x, y);
    }
} // button down, operator tool selected?
} else if (strcmp(args[0], "motion") == 0) { // button not down
    if (tool_state == SELECT_TOOL) {
        if (selected_object_ptr != NULL) {
            drawing_changed = true;
            if (text_selected) {
                if (first_draw == true) {
                    shadow_width = selected_object_ptr->text_width();
                    shadow_height = selected_object_ptr->text_height();
                    draw_text_shadow(x, y, shadow_width, shadow_height);
                    first_draw = false;
                } // first_draw
            } else {
                draw_text_shadow(x_state, y_state, shadow_width,
shadow_height);
                draw_text_shadow(x, y, shadow_width, shadow_height);
            }
        } // text_selected
    } else
        if (selected_object_ptr->is_a() == OPERATOROBJECT) {
            if (handle_selected == true) {

                if (first_draw == true) {
                    selected_object_ptr->erase();
                    selected_object_ptr->unselect();
                    selected_object_ptr->draw(SOLID);
                    temp_operator_ptr->draw(DOTTED);
                    first_draw = false;
                } // first_draw
                else {
                    temp_operator_ptr->move_handle(x - x_state,
                                                    y - y_state);
                }
            } // handle_selected
        } else {
            if (first_draw == true)

// Drawing the same thing twice in xor mode erases it. When
// moving an object, it is drawn once the first and last time,
// and twice afterwards

                first_draw = false;
            else
                temp_operator_ptr->draw(DOTTED);
            temp_operator_ptr->move(x - x_state, y - y_state);
            temp_operator_ptr->draw(DOTTED);
        }
    } // is_a OPERATOROBJECT
} else {
    if (selected_object_ptr->is_a() == STREAMOBJECT) {
        if (handle_selected) {
            if (first_draw == true) {
                conv_st_ptr =
                    (StreamObject *) selected_object_ptr;
                conv_st_ptr->erase_handle();
                draw_handle(std_graphics_context, x, y);
                first_draw = false;
            } // first_draw
        } else {

```

```

        draw_handle(std_graphics_context, x_state, y_state);
        draw_handle(std_graphics_context, x, y);
        selected_object_ptr->move_handle(x - x_state,
                                         y - y_state);
    }
    } // handle_selected
} // is_a STREAMOBJECT

x_state = x;
y_state = y;
} // selected_object_ptr != NULL
} // tool_state == SELECT_TOOL

// I-bar mode check
temp_object_ptr = graphic_list.over(x, y);
if (temp_object_ptr != NULL) {
    object_def = temp_object_ptr->is_a();
    if (object_def == OPERATOROBJECT) {

#ifdef GE_DEBUG
//          cerr << "It is an Operator Object" << endl;
#endif /* GE_DEBUG */

        ibar_mode = true;
        setcursor(drawing_a, True, XC_xterm);
    } // object_def == OPERATOROBJECT
    else
        if (object_def == STREAMOBJECT) {

#ifdef GE_DEBUG
//          cerr << "It is an Stream Object" << endl;
#endif /* GE_DEBUG */

            ibar_mode = true;
            setcursor(drawing_a, True, XC_xterm);
        } // object_def == STREAMOBJECT
        else {
            ibar_mode = false;
            setcursor(drawing_a, False, None);
        }
    } // temp_object_ptr != NULL
    else { // No object selected

#ifdef GE_DEBUG
//          cerr << "No object selected Object" << endl;
#endif /* GE_DEBUG */

        ibar_mode = false;
        setcursor(drawing_a, False, None);
    } // No object selected
} else if (strcmp(args[0], "up") == 0) {
    if (tool_state == SELECT_TOOL)
        if (selected_object_ptr != NULL) {
            if (text_selected) {
                if (first_draw == false) {
                    draw_text_shadow(x_state, y_state,
                                     shadow_width, shadow_height);
                    selected_object_ptr->text_locate(x, y);
                    save_state(SAVE_REQUIRED);
                } // first_draw
            }
        }
    }
}

```

```

    } // text_selected
    else
        if (selected_object_ptr->is_a() == OPERATOROBJECT) {
            if (first_draw == false) {
                temp_operator_ptr->draw(DOTTED);
                XYPAIR temp_pair = temp_operator_ptr->center();
                conv_op_ptr =
                    (OperatorObject *) selected_object_ptr;
                conv_op_ptr->radius(temp_operator_ptr->radius());
                conv_op_ptr->set_location(temp_pair.x,
                                        temp_pair.y);
                if (handle_selected)
                    conv_op_ptr->set_default_text_location();
            } // first draw
        } // is_a OPERATOROBJECT
        else
            if ((selected_object_ptr->is_a() == STREAMOBJECT)
                && (handle_selected)) {
                draw_handle(std_graphics_context, x_state, y_state);
            } // is_a STREAMOBJECT
        if (drawing_changed == true) {
            graphic_list.move_notify(selected_object_ptr->is_a(),
                                    selected_object_ptr->id());
            graphic_list.draw();
            save_state(SAVE_REQUIRED);
            drawing_changed = false;
        } // drawing_changed
        handle_selected = false;
        } // selected_object_ptr != NULL
    first_draw = true;
} else if (strcmp(args[0], "btn3down") == 0) {
    clear_status();
    if (ibar_mode == true) {
        if (object_def == OPERATOROBJECT) {
            // op_being_updated = (OperatorObject *) temp_object_ptr;
            operator_property_dialog(drawing_a, op_being_updated, x, y,
                                    graphic_list.cur_op_is_terminator(),
                                    graphic_list.avail_impl_langs_adr(),
                                    &graphic_list);
        }
        else
            if (object_def == STREAMOBJECT) {
                stream_property_dialog(drawing_a, st_being_updated, x, y,
                                       &graphic_list);
                // XFlush(XtDisplay(drawing_a)); /* Stub for stream code */
            }
        else {
            XFlush(XtDisplay(drawing_a)); /* Stub for Non Operator of Stream
*/
        }
    } // ibar_mode
} else if (strcmp(args[0], "btn3motion") == 0) {
    temp_object_ptr = graphic_list.over(x, y);
    if (temp_object_ptr != NULL) {
        object_def = temp_object_ptr->is_a();
        if (object_def == OPERATOROBJECT) {
#ifdef GE_DEBUG
            // cerr << "It is an Operator Object" << endl;
#endif /* GE_DEBUG */

```

```

    ibar_mode = true;
    setcursor(drawing_a, True, XC_xterm);

    op_being_updated = (OperatorObject *) temp_object_ptr;
}
else
    if (object_def == STREAMOBJECT) {

#ifdef GE_DEBUG
    //          cerr << "It is an Stream Object" << endl;
#endif /* GE_DEBUG */

    ibar_mode = true;
    setcursor(drawing_a, True, XC_xterm);

    st_being_updated = (StreamObject *) temp_object_ptr;

}
else {
    ibar_mode = false;
    setcursor(drawing_a, True, XC_left_ptr);
}
}
else { // No object selected

#ifdef GE_DEBUG
    cerr << "No object selected Object" << endl;
#endif /* GE_DEBUG */

    ibar_mode = false;
    setcursor(drawing_a, False, None);
}
} else if (strcmp(args[0], "btn3up") == 0) {
    XFlush(XtDisplay(drawing_a)); /* Stub for stream code */
} else if (strcmp(args[0], "motionnotify") == 0) {
    if (label_edit_mode == true) {
        label_edit_mode = false;
        if (ibar_object_ptr) {
            if (ibar_object_ptr->is_a() == STREAMOBJECT) {
                labelName = ((StreamObject *)ibar_object_ptr)->name();

                // MY
                if ( is_empty(labelName) )
                {
                    sprintf(default_name, "noname_%d", get_unique_id());
                    labelName = default_name;
                    ((StreamObject *)ibar_object_ptr)->name(labelName);
                    ((StreamObject *)ibar_object_ptr)->draw(SOLID);
                }

                warningMSG = (char *) malloc(strlen(labelName)+40);
                if (!valid_id(labelName)) {
                    sprintf(warningMSG, "Invalid stream name: %s", labelName);
                    warning(drawing_a, warningMSG);
                    update_status(
                        "Illegal stream name, retype: id ::= letter
{alpha_numeric}",
                        RING_BELL);
                    ((StreamObject *)ibar_object_ptr)->erase_text();

                // MY

```

```

        sprintf(default_name, "noname_%d", get_unique_id());
        ((StreamObject *)ibar_object_ptr)->name(default_name);
        //((StreamObject *)ibar_object_ptr)->name("");

        ((StreamObject *)ibar_object_ptr)->draw_text(SOLID);
    } else if (is_keyword(labelName, false)) {
        sprintf(warningMSG, "Stream name is a keyword: %s", labelName);
        warning(drawing_a, warningMSG);
        update_status("Stream name is a keyword, retype", RING_BELL);
        ((StreamObject *)ibar_object_ptr)->erase_text();

        // MY
        sprintf(default_name, "noname_%d", get_unique_id());
        ((StreamObject *)ibar_object_ptr)->name(default_name);
        //((StreamObject *)ibar_object_ptr)->name("");

        ((StreamObject *)ibar_object_ptr)->draw_text(SOLID);
    } else {
        // Valid stream name, get any existing type information
        type_match = graphic_list.fetch_matching_stream_type(
            (StreamObject *)ibar_object_ptr, &state_change);
        if (state_change)
            ((StreamObject *)ibar_object_ptr)->draw(SOLID);
    }
    free(labelName);
    free(warningMSG);
}
else {
    labelName = ((OperatorObject *)ibar_object_ptr)->name();
    type_operator = (strchr(labelName, '.')) ? true : false;

    // MY
    if (is_empty(labelName) )
    {
        sprintf(default_name, "noname_%d", get_unique_id());
        labelName = default_name;
        ((OperatorObject *)ibar_object_ptr)->name(labelName);
        ((OperatorObject *)ibar_object_ptr)->draw(SOLID);
    }

    warningMSG = (char *) malloc(strlen(labelName)+80);

    if (!valid_op_id(labelName)) {
        sprintf(warningMSG,
            "Invalid operator name (syntax or keyword): %s", labelName);
        warning(drawing_a, warningMSG);
        update_status("Illegal operator name, retype:"
            " op_id ::= [id '.'] op_name ['(' [id_list] '|' [id_list]
            ') ']",
            RING_BELL);
        ((OperatorObject *)ibar_object_ptr)->erase_text();

        // MY
        sprintf(default_name, "noname_%d", get_unique_id());
        ((OperatorObject *)ibar_object_ptr)->name(default_name);
        //((OperatorObject *)ibar_object_ptr)->name("");

        ((OperatorObject *)ibar_object_ptr)->draw_text(SOLID);
    } else if (type_operator &&
        (((OperatorObject *)ibar_object_ptr)->is_composite()))
    {

```

```

sprintf(warningMSG,
        "A Composite Operator can not be a Type: %s", labelName);
warning(drawing_a, warningMSG);
update_status("Composite Operator can not be a Type:"
              " rename operator or make Automic",
              RING_BELL);
((OperatorObject *)ibar_object_ptr)->erase_text();

// MY
    sprintf(default_name, "noname_%d", get_unique_id());
    ((OperatorObject *)ibar_object_ptr)->name(default_name);
    //((OperatorObject *)ibar_object_ptr)->name("");

    ((OperatorObject *)ibar_object_ptr)->draw_text(SOLID);
} else if (!type_operator &&
!graphic_list.unique_op_id(labelName,
    ((OperatorObject *)ibar_object_ptr)->id())) {
    sprintf(warningMSG,
        "Simple Operator Names must be unique to level: %s",
labelName);
    warning(drawing_a, warningMSG);
    update_status("Operators that are not types must have a "
                  "unique name",
                  RING_BELL);
    ((OperatorObject *)ibar_object_ptr)->erase_text();

    // MY
        sprintf(default_name, "noname_%d", get_unique_id());
        ((OperatorObject *)ibar_object_ptr)->name(default_name);
        //((OperatorObject *)ibar_object_ptr)->name("");

        ((OperatorObject *)ibar_object_ptr)->draw_text(SOLID);
    }
    free(labelName);
}
}
string[0] = NULL;
buffer[0] = NULL;
ibar_object_ptr = NULL;
}
temp_object_ptr = graphic_list.over(x, y);
if (temp_object_ptr != NULL) {
    object_def = temp_object_ptr->is_a();
    if (object_def == OPERATOROBJECT) {

#ifdef GE_DEBUG
//          cerr << "It is an Operator Object" << endl;
#endif /* GE_DEBUG */

        ibar_mode = true;
        setcursor(drawing_a, True, XC_xterm);

        op_being_updated = (OperatorObject *) temp_object_ptr;
    }
    else
        if (object_def == STREAMOBJECT) {

#ifdef GE_DEBUG
//          cerr << "It is an Stream Object" << endl;
#endif /* GE_DEBUG */

```



```

        ibar_mode = true;
        setcursor(drawing_a, True, XC_xterm);

        st_being_updated = (StreamObject *) temp_object_ptr;
    }
    else {
        ibar_mode = false;
        setcursor(drawing_a, True, XC_left_ptr);
    }
}
else { // No object selected

#ifdef GE_DEBUG
//          cerr << "No object selected Object" << endl;
#endif /* GE_DEBUG */

        ibar_mode = false;
        setcursor(drawing_a, False, None);
    }
    else if (strcmp(args[0], "key") == 0) {

#ifdef GE_DEBUG
//          cout << "key pressed: " << event->xkey.keycode << endl;
#endif

        count = XLookupString(&event->xkey, buffer,
                               bufsize, &keysym, NULL);
        buffer[count] = NULL; /* add NULL terminator */

        if (label_edit_mode==true) {
            if ((keysym == XK_Return) || (keysym == XK_KP_Enter) ||
                (keysym == XK_Linefeed)) {
                label_edit_mode = false;
                if (ibar_object_ptr) {
                    if (ibar_object_ptr->is_a() == STREAMOBJECT) {
                        labelName = ((StreamObject *)ibar_object_ptr)->name();
                        warningMSG = (char *) malloc(strlen(labelName)+40);
                        if (!valid_id(labelName)) {
                            sprintf(warningMSG, "Invalid stream name: %s", labelName);
                            warning(drawing_a, warningMSG);
                            update_status(
                                "Illegal stream name, retype: id ::= letter
{alpha_numeric}",
                                RING_BELL);
                            ((StreamObject *)ibar_object_ptr)->erase_text();
                            ((StreamObject *)ibar_object_ptr)->name("");
                            ((StreamObject *)ibar_object_ptr)->draw_text(SOLID);
                        } else if (is_keyword(labelName, false)) {
                            sprintf(warningMSG, "Stream name is a keyword: %s",
labelName);
warning(drawing_a, warningMSG);
update_status("Stream name is a keyword, retype",
RING_BELL);
                            ((StreamObject *)ibar_object_ptr)->erase_text();
                            ((StreamObject *)ibar_object_ptr)->name("");
                            ((StreamObject *)ibar_object_ptr)->draw_text(SOLID);
                        } else {
                            // Valid stream name, get any existing type information
                            type_match = graphic_list.fetch_matching_stream_type(
                                (StreamObject *)ibar_object_ptr, &state_change);
                            if (state_change)

```

```

        ((StreamObject *)ibar_object_ptr)->draw(SOLID);
    }
    free(labelName);
    free(warningMSG);
}
else {
    labelName = ((OperatorObject *)ibar_object_ptr)->name();
    type_operator = (strchr(labelName, '.')) ? true : false;

    warningMSG = (char *) malloc(strlen(labelName)+80);
    if (!valid_op_id(labelName)) {
        sprintf(warningMSG,
            "Invalid operator name (syntax or keyword): %s",
labelName);
        warning(drawing_a, warningMSG);
        update_status("Illegal operator name, retype: "
            "op_id ::= [id '.'] op_name ['(' [id_list] '|' [id_list]
            ')'],
            RING_BELL);
        ((OperatorObject *)ibar_object_ptr)->erase_text();
        ((OperatorObject *)ibar_object_ptr)->name("");
        ((OperatorObject *)ibar_object_ptr)->draw_text(SOLID);
    } else if ((strchr(labelName, '.') != NULL) &&
        (((OperatorObject *)ibar_object_ptr)-
>is_composite())) {
        sprintf(warningMSG,
            "A Composite Operator can not be a Type: %s",
labelName);
        warning(drawing_a, warningMSG);
        update_status("Composite Operator can not be a Type:"
            " rename operator or make Automic",
            RING_BELL);
        ((OperatorObject *)ibar_object_ptr)->erase_text();
        ((OperatorObject *)ibar_object_ptr)->name("");
        ((OperatorObject *)ibar_object_ptr)->draw_text(SOLID);
    } else if (!type_operator &&
!graphic_list.unique_op_id(labelName,
        ((OperatorObject *)ibar_object_ptr)->id())) {
        sprintf(warningMSG,
            "Simple Operator Names must be unique to level: %s",
labelName);
        warning(drawing_a, warningMSG);
        update_status("Operators that are not types must have a "
            "unique name",
            RING_BELL);
        ((OperatorObject *)ibar_object_ptr)->erase_text();
        ((OperatorObject *)ibar_object_ptr)->name("");
        ((OperatorObject *)ibar_object_ptr)->draw_text(SOLID);
    }
    free(labelName);
}
}
string[0] = NULL;
buffer[0] = NULL;
ibar_object_ptr = NULL;
}
else
if (((keySYM >= XK_KP_Space) && (keySYM <= XK_KP_9)) ||
    ((keySYM >= XK_Space) && (keySYM <= XK_asciitilde))) {
    if ((strlen(string) + strlen(buffer)) >= INPUT_LINE_SIZE) {
        XBell(display_ptr, 100);
    }
}

```

```

    }
    else {
        strcat(string, buffer);
    }
}
else
    if ((keysym >= XK_Shift_L) && (keysym <= XK_Hyper_R)) {
        ; /* Do nothing because it's a modifier key */
    }
    else
        if ((keysym >= XK_F1) && (keysym <= XK_F35)) {
            if (buffer[0] != (char)NULL) {
                if ((strlen(string) + strlen(buffer)) >= INPUT_LINE_SIZE) {
                    XBell(display_ptr, 100);
                }
                else {
                    strcat(string, buffer);
                }
            }
        }
        else
            if ((keysym == XK_BackSpace) ||
                (keysym == XK_Delete)) {
                if ((length = strlen(string)) > 0) {
                    string[length - 1] = NULL;
                }
                else {
                    XBell(display_ptr, 100);
                }
            }

temp_object_ptr = graphic_list.over(x, y);
if (temp_object_ptr != NULL) {
    object_def = temp_object_ptr->is_a();
    if (label_edit_mode != false &&
        (object_def == OPERATOROBJECT ||
         object_def == STREAMOBJECT)) {

        ibar_object_ptr = temp_object_ptr;
        temp_object_ptr->erase_text();
        temp_object_ptr->name(string);
        temp_object_ptr->draw_text(SOLID);
        save_state(SAVE_REQUIRED);
        temp_object_ptr->unselect();
    }
}
}
else
    if (alt_selected || ctrl_selected) {           // alt key pressed
        alt_selected = false;
        ctrl_selected = false;
        switch(keysym) {
            case XK_D:                               // Decompose
            case XK_d:
                handle_psd_options(2);               // MY: 3 -> 2
                break;
            case XK_P:                               // Goto Parent
            case XK_p:
                handle_psd_options(1);               // MY: 2 -> 1
                break;
            case XK_R:                               // Goto Root

```

```

case XK_r:
    handle_psd_options(0);      // MY: 1 -> 0
    break;
case XK_F:                      // Refresh Display
case XK_f:
    handle_edit_options(4);
    break;
case XK_Meta_L:                 // Alt key to activate
case XK_Meta_R:
    alt_selected = true;
    break;
case XK_Control_L:              // Control key to activate
case XK_Control_R:
    ctrl_selected = true;
    break;
default:
    break;
}
}
else if (keysym == XK_Meta_L || keysym == XK_Meta_R) {
    alt_selected = true;
}
else if (keysym == XK_Control_L || keysym == XK_Control_R) {
    ctrl_selected = true;
}
else
if (selected_object_ptr != NULL) {
    if ((keysym == XK_BackSpace) ||
        (keysym == XK_Delete)) {
        selected_object_ptr->erase();
        OP_ID deleted_op_id = selected_object_ptr->id();
        save_state(SAVE_REQUIRED);
        graphic_list.delete_notify(selected_object_ptr->
                                   is_a(), deleted_op_id);
        selected_object_ptr->set_deleted();
        selected_object_ptr = NULL;
        graphic_list.draw();
        ibar_mode = false;
        setcursor(drawing_a, False, None);
    }
}
else
if (ibar_mode==true &&
    label_edit_mode==false &&
    (((keysym >= XK_KP_Space) && (keysym <= XK_KP_9)) ||
     ((keysym >= XK_space) && (keysym <= XK_asciitilde)))) {
    if ((strlen(string) + strlen(buffer)) >= INPUT_LINE_SIZE) {
        XBell(display_ptr, 100);
    }
    else {
        strcat(string, buffer);
    }

    label_edit_mode = true;
    clear_status();

    temp_object_ptr = graphic_list.over(x, y);
    if (temp_object_ptr != NULL) {
        object_def = temp_object_ptr->is_a();
        if (object_def == OPERATOROBJECT ||
            object_def == STREAMOBJECT) {

```

```

        //          temp_object_ptr->select();
        ibar_object_ptr = temp_object_ptr;
        temp_object_ptr->erase_text();
        temp_object_ptr->name(string);
        temp_object_ptr->draw_text(SOLID);
        save_state(SAVE_REQUIRED);
//          temp_object_ptr->unselect();
        } // label_edit_mode != false && ()
        } // temp_object_ptr != NULL
        } // ibar_mode && label_edit_mode == false && ()
    } // strcmp KEY
} // draw

// Callback function. Just destroys the widget.
void widget_killer(Widget widget, XtPointer, XtPointer) {
    XtDestroyWidget(widget);
}

// Callback function. Called when Operator Tool button is
// pressed.
void op_button_cb(Widget, XtPointer, XtPointer) {
    select_state(OPERATOR_TOOL);

    XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);
    if (selected_object_ptr != NULL) {
        selected_object_ptr->unselect();
        selected_object_ptr = NULL;
    }
    //?? XtVaSetValues(tool_indicator, XmNvalue, "Operator Tool", NULL);
}

// Callback function. Called when Terminator Tool button is
// pressed.
void term_button_cb(Widget, XtPointer, XtPointer) {
    select_state(TERMINATOR_TOOL);

    XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);
    if (selected_object_ptr != NULL) {
        selected_object_ptr->unselect();
        selected_object_ptr = NULL;
    }
    //?? XtVaSetValues(tool_indicator, XmNvalue, "Terminator Tool",
    NULL);
}

// Null Callback.
void null_cb(Widget, XtPointer, XtPointer) {}

// Callback function. Called when Timer Tool OK button is
// pressed. DL 8/22/96; KBM 10/24/96

```

```

void timer_tool_ok_cb(Widget parent, XtPointer client_data,
                      XtPointer call_data) {

    Widget          list_w = (Widget)client_data;
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *)call_data;
    int             u_bound;
    XmString        *strlist;
    char            *text;
    ID_LIST         op, tp;
    ID_LIST         otimer = graphic_list.timer_list(); // MY
    ID_LIST         idp, timers;

    enter_timer = 0; // MY 8/5/97

    XtVaGetValues(list_w,
                  XmNitemCount,    &u_bound,
                  XmNitems,        &strlist,
                  NULL);

    timers = NULL;
    if (u_bound > 0) {
        idp = (ID_LIST) malloc(sizeof(ID_NODE));
        idp->next = NULL;
        //if (XmStringGetLtoR(strlist[0], XmFONTLIST_DEFAULT_TAG, &text))
    //@1
        if (XmStringGetLtoR(strlist[0], XmSTRING_DEFAULT_CHARSET, &text))
    //@1
            idp->id = text;
        timers = idp;

        for (int i = 1; i < u_bound; i++) {
            idp->next = (ID_LIST) malloc(sizeof(ID_NODE));
            idp = idp->next;
            idp->next = NULL;
            //if (XmStringGetLtoR(strlist[i], XmFONTLIST_DEFAULT_TAG,
    &text))//@1
            if (XmStringGetLtoR(strlist[i], XmSTRING_DEFAULT_CHARSET,
    &text))//@1
                idp->id = text;
        }
    }

    // MY
    op = otimer; tp = timers;
    while ( op != NULL && tp != NULL )
    {
        if ( strcmp(op->id, tp->id) != 0 ) {
            save_state(SAVE_REQUIRED);
            break;
        }
        op = op->next; tp = tp->next;
    }
    if ( op != NULL ) save_state(SAVE_REQUIRED);
    if ( tp != NULL ) save_state(SAVE_REQUIRED);

    graphic_list.timer_list(timers);
    id_list_release(timers);      timers = NULL;

    XtDestroyWidget(XtParent(XtParent(XtParent(parent))));
}

```

```

}

//      Callback function.  Called when Timer Tool button is
// pressed.  DL 8/16/96.

void timers_button_cb(Widget parent,XtPointer client_data,XtPointer
call_data){

    Widget    dialog, rc, pane, list, action_a;
    int        count = 0, i, n=0;
    ID_LIST    idp, timers;
    Arg        args[5];
    XmString *str,string;
    static ActionAreaItem action_items[] = {
        {"OK",      timer_tool_ok_cb,  NULL},
        {"Cancel",  timer_close_dialog, NULL},
        {"Add",     timer_tool_add_cb,  NULL},
        {"Delete",  timer_tool_del_cb,  NULL},
        {"Edit",    timer_tool_edit_cb, NULL},
        {"Help",    help_cb,            "timers_tool.hlp"}
    };

    // MY 8/5/97
    if ( enter_timer == 1 ) {
        putchar(007);
        return;
    }
    if ( enter_errs == 1 ) {
        putchar(007);
        warning(parent, "Please close error message window");
        return;
    }
    enter_timer = 1;

    //Build list for list widget
    timers = graphic_list.timer_list();

    idp = timers;
    while(idp) {
        count++;
        idp = idp->next;
    }
    idp = timers;
    str = (XmString *) XtMalloc (count * sizeof (XmString));
    for (i = 0; i < count; i++) {
        // str[i] = XmStringCreateLocalized(idp->id);      // @1
        str[i] = XmStringCreateSimple(idp->id);          // @1
        idp = idp->next;
    }
    id_list_release(timers);      timers = NULL;

    dialog = XtVaCreatePopupShell("dialog", xmDialogShellWidgetClass,
                                  XtParent(parent), XmNtitle, "Timers Tool",
                                  XmNdeleteResponse, XmDESTROY,
                                  NULL);

    action_items[1].data = (XtPointer)dialog; //Set cancel buttons
    client_data

```

```

pane = XtVaCreateWidget("pane", xmPanedWindowWidgetClass, dialog,
                        XmNsashWidth, 1,
                        XmNsashHeight, 1,
                        NULL);

rc = XtVaCreateWidget("control_area", xmRowColumnWidgetClass, pane,
NULL);
// string = XmStringCreateLocalized("Enter or Edit Timers"); // @1
string = XmStringCreateSimple("Enter or Edit Timers"); // @1
XtVaCreateManagedWidget("label", xmLabelGadgetClass, rc,
                        XmNlabelString, string,
                        NULL);
XmStringFree(string);

list = XmCreateScrolledList(rc, "Timer_List", NULL, 0);
XtVaSetValues(list,
                XmNvisibleItemCount, 10,
                XmNitemCount, count,
                XmNitems, str,
                NULL);
XtManageChild(list);
for(i = 0; i < count; i++)
    XmStringFree(str[i]);

XtManageChild(rc);

//Set client data for "OK", "Add", "Del", and "Edit" buttons
action_items[0].data = (XtPointer)list;
action_items[2].data = (XtPointer)list;
action_items[3].data = (XtPointer)list;
action_items[4].data = (XtPointer)list;

action_a = CreateActionArea(pane, action_items,
XtNumber(action_items));

XtManageChild(pane);
XtPopup(dialog, XtGrabNone);
}

// Callback function. Called when Informal Description Tool OK is
// pressed. Added by Doug Lange 8/19/96.
static void inform_tool_ok_pushed(Widget w, XtPointer client_data,
                                XtPointer call_data) {

Widget text_w = (Widget)client_data;
XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *)call_data;

char *text = XmTextGetString(text_w);
char *org_text = graphic_list.graph_informal_desc();

enter_inform = 0; // MY 8/5/97

// MY
if (strcmp(text, org_text) != 0) {
    graphic_list.graph_informal_desc(text);
    save_state(SAVE_REQUIRED);
}
free(text);

```



```

XtDestroyWidget(XtParent(XtParent(XtParent(w))));

clear_status();
}

//      Callback function.  Called when Informal Description Tool Button
is
// pressed.  Added by Doug Lange 8/19/96.

static void informal_button_cb(Widget w, XtPointer client_data,
                               XtPointer call_data)
{
    Widget      dialog, pane, rc, text_w, action_a;
    XmString    string;
    char        *description;

    static ActionAreaItem  action_items[] = {
        {"OK",      inform_tool_ok_pushed, NULL,          },
        {"Cancel",  inform_close_dialog,   NULL,          },
        {"Help",    help_cb,               "inform_tool.hlp" }
    };

    // MY 8/5/97
    if ( enter_inform == 1 ) {
        putchar(007);
        return;
    }
    if ( enter_errs == 1 ) {
        putchar(007);
        warning(w, "Please close error message window");
        return;
    }
    enter_inform = 1;

    dialog = XtVaCreatePopupShell ("dialog", xmDialogShellWidgetClass,
                                   XtParent(w),
                                   XmNtitle, "Informal Design Description",
                                   XmNdeleteResponse, XmDESTROY,
                                   NULL);

    action_items[1].data = (XtPointer)dialog;  //Set cancel buttons
    client_data

    pane = XtVaCreateWidget("pane", xmPanedWindowWidgetClass, dialog,
                           XmNsashWidth, 1,
                           XmNsashHeight, 1,
                           NULL);

    rc = XtVaCreateWidget("control_area", xmRowColumnWidgetClass, pane,
                           NULL);
    string = XmStringCreateSimple("Enter or Edit Informal Description");
    //@1
    XtVaCreateManagedWidget("label", xmLabelGadgetClass, rc,
                             XmNlabelString, string,
                             NULL);
    XmStringFree(string);

    description = graphic_list.graph_informal_desc();

```

```

int      n = 0;
Arg      args[10];
XtSetArg(args[n], XmNrows,          12); n++;
XtSetArg(args[n], XmNcolumns,       70); n++;
XtSetArg(args[n], XmNscrollVertical, true); n++;
XtSetArg(args[n], XmNscrollHorizontal, false); n++;
XtSetArg(args[n], XmNeditMode,      XmMULTI_LINE_EDIT); n++;
XtSetArg(args[n], XmNeditable,      true); n++;
XtSetArg(args[n], XmNcursorPositionVisible, true); n++;
XtSetArg(args[n], XmNwordWrap,      true); n++;
XtSetArg(args[n], XmNvalue,         description); n++;
text_w = XmCreateScrolledText(rc, "text-field", args, n);
XtManageChild(text_w);
//text_w = XtVaCreateManagedWidget("text-field",
xmTextFieldWidgetClass,
//      rc, NULL);

XtAddCallback(text_w, XmNmodifyVerifyCallback, validate_text, NULL);

XtManageChild(rc);

//Set client data for the "OK" and "Cancel" buttons
action_items[0].data = (XtPointer)text_w;

action_a = CreateActionArea(pane, action_items,
XtNumber(action_items));

//XtAddCallback(text_w, XmNactivateCallback, activate_cb, action_a);

XtManageChild(pane);
free(description);
XtPopup(dialog, XtGrabNone);
}

// Callback function. Called when Stream Tool button is
// pressed.
void stream_button_cb(Widget, XtPointer, XtPointer) {
    select_state(STREAM_TOOL);

    XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);
    if (selected_object_ptr != NULL) {
        selected_object_ptr->unselect();
        selected_object_ptr = NULL;
    }
    //?? XtVaSetValues(tool_indicator, XmNvalue, "Stream Tool", NULL);
}

// Callback function. Called when Select Tool button is
// pressed.
void select_button_cb(Widget, XtPointer, XtPointer) {
    select_state(SELECT_TOOL);

    XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);
    //?? XtVaSetValues(tool_indicator, XmNvalue, "Select Tool", NULL);
}

```

```

static void types_tool_ok_pushed(Widget w, XtPointer client_data,
                                XtPointer call_data) {

    Widget text_w = (Widget)client_data;
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *)call_data;

    char *text = XmTextGetString(text_w);
    char *org_text = graphic_list.global_types();

    enter_types = 0; // MY 8/5/97

    /*
     * MY
     */

    {
        int ok, error_line, error_column, error_token_length;
        long l;
        XmString label;

        parse_type_spec(text, &ok, &error_line,
&error_column,&error_token_length);

        if (ok)
            printf("parser return ok\n");
        else
        {
            printf("parser return NOT ok\n");
            printf("error line   = %d\n", error_line);
            printf("error column = %d\n", error_column);
            printf("error length = %d\n", error_token_length);
            putchar(007); putchar(007); putchar(007);
            l = XmTextGetCursorPosition(text_w);
            l = (long) error_line - 1L;
            XmTextSetSelection(text_w, l, l+(long)error_token_length,
(Time)10);
            XmTextSetCursorPosition(text_w, l);

            return;
        }
    }

    /*
     * MY
     */

    if (strcmp(text, org_text) != 0) {
        graphic_list.global_types(text);
        save_state(SAVE_REQUIRED);
    }

    free(text);                text      = NULL;
    free(org_text);            org_text = NULL;

    XtDestroyWidget(XtParent(XtParent(XtParent(w))));

    clear_status();
}

```

```

void types_button_cb(Widget w, XtPointer client_data,
                    XtPointer call_data) {

    Widget    dialog, pane, rc, text_w, action_a;
    XmString  string;
    char      *description;

    static ActionAreaItem  action_items[] = {
        {"OK",      types_tool_ok_pushed,  NULL           },
        {"Cancel",  types_close_dialog,    NULL           },
        {"Help",    help_cb,               "types_tool.hlp" }
    };

    // MY 8/5/97
    if ( enter_types == 1 ) {
        putchar(007);
        return;
    }
    if ( enter_errs == 1 ) {
        putchar(007);
        warning(w, "Please close error message window");
        return;
    }
    enter_types = 1;

    dialog = XtVaCreatePopupShell ("dialog", xmDialogShellWidgetClass,
                                   XtParent(w),
                                   XmNtitle, "Prototype Types Specification",
                                   XmNdeleteResponse, XmDESTROY,
                                   NULL);

    action_items[1].data = (XtPointer)dialog; //Set cancel buttons
    client_data

    pane = XtVaCreateWidget("pane", xmPanedWindowWidgetClass, dialog,
                           XmNsashWidth, 1,
                           XmNsashHeight, 1,
                           NULL);

    rc = XtVaCreateWidget("control_area", xmRowColumnWidgetClass, pane,
                          NULL);
    string = XmStringCreateSimple("View or Edit Prototype Types
Specification");
    XtVaCreateManagedWidget("label", xmLabelGadgetClass, rc,
                           XmNlabelString, string,
                           NULL);
    XmStringFree(string);

    description = graphic_list.global_types();

    int      n = 0;
    Arg      args[10];
    XtSetArg(args[n], XmNrows, 12); n++;
    XtSetArg(args[n], XmNcolumns, 70); n++;
    XtSetArg(args[n], XmNscrollVertical, true); n++;
    XtSetArg(args[n], XmNscrollHorizontal, true); n++;
    XtSetArg(args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
    XtSetArg(args[n], XmNeditable, true); n++;
    XtSetArg(args[n], XmNcursorPositionVisible, true); n++;
    XtSetArg(args[n], XmNwordWrap, true); n++;
    XtSetArg(args[n], XmNvalue, description); n++;

```

```

    text_w = XmCreateScrolledText(rc, "text-field", args, n);
    XtManageChild(text_w);
    //text_w = XtVaCreateManagedWidget("text-field",
xmTextFieldWidgetClass,
    //      rc, NULL);

    // XtAddCallback(text_w, XmNmodifyVerifyCallback, validate_text,
NULL);
    // Note: If you have problems with '}' symbols in the text,
uncomment
    // the line above.

    XtManageChild(rc);

    //Set client data for the "OK" and "Cancel" buttons
    action_items[0].data = (XtPointer)text_w;

    action_a = CreateActionArea(pane, action_items,
XtNumber(action_items));

    //XtAddCallback(text_w, XmNactivateCallback, activate_cb, action_a);

    XtManageChild(pane);

/*
 * MY
 */

{
    GRAPH_DESC tmp_gd = gdnode;
    char buffer[100];
    char type_name[100];
    char type_buffer[5000];

    strcpy(type_buffer, "");
    if (strcmp(XmTextGetString(text_w), "") == 0 ||
        strcmp(XmTextGetString(text_w), "\n") == 0 )
        while ( tmp_gd->stream_list != NULL )
        {
            sprintf(type_name, "%s#", tmp_gd->stream_list->st-
>stream_type_name);
            if ( strstr(type_buffer, type_name) == NULL && strcmp(type_name,
"##") )
            {
                sprintf(buffer, "TYPE %s\nSPECIFICATION\nEND\nIMPLEMENTATION ADA
%s END\n\n",
                    tmp_gd->stream_list->st->stream_type_name,
                    tmp_gd->stream_list->st->stream_type_name);
                XmTextInsert (text_w, 0L, buffer);
                strcat(type_buffer, "#");
                strcat(type_buffer, tmp_gd->stream_list->st->stream_type_name);
                strcat(type_buffer, "#");
            }
            tmp_gd->stream_list = tmp_gd->stream_list->next;
        }
}

/*
 * MY
 */

free(description);

```

```

XtPopup(dialog, XtGrabNone);
}

static void spec_tool_ok_pushed(Widget w, XtPointer client_data,
                                XtPointer call_data) {

    Widget text_w = (Widget)client_data;
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct *)call_data;

    char *text = XmTextGetString(text_w);
    char *org_text = graphic_list.cur_op_spec();

    enter_spec = 0; // MY 8/5/97

    /*
     * MY
     */

    {

        int ok, error_line, error_column, error_token_length;
        long l;
        XmString label;

        parse_oper_spec(text, &ok, &error_line,
                        &error_column, &error_token_length);

        if (ok)
            printf("parser return ok\n");
        else
        {
            printf("parser return NOT ok\n");
            printf("error line   = %d\n", error_line);
            printf("error column = %d\n", error_column);
            printf("error length = %d\n", error_token_length);
            putchar(007); putchar(007); putchar(007);

            l = XmTextGetCursorPosition(text_w);
            l = (long) error_line - 1L;
            XmTextSetSelection(text_w, l, l+(long)error_token_length,
(Time)10);
            XmTextSetCursorPosition(text_w, l);

            return;
        }

    }

    /*
     * MY
     */

    if (strcmp(text, org_text) != 0) {
        graphic_list.cur_op_spec(text);
        save_state(SAVE_REQUIRED);
    }

    free(text);
    free(org_text);
    text = NULL;
    org_text = NULL;
}

```

```

XtDestroyWidget(XtParent(XtParent(XtParent(w))));

clear_status();
}

void spec_button_cb(Widget w, XtPointer client_data,
                   XtPointer call_data)
{
    Widget      dialog, pane, rc, text_w, action_a;
    XmString    string;
    char        *description;

    static ActionAreaItem  action_items[] = {
        {"OK",          spec_tool_ok_pushed, NULL,          },
        {"Cancel",      spec_close_dialog,   NULL,          },
        {"Help",        help_cb,             "spec_tool.hlp" }
    };

    // MY 8/5/97
    if ( enter_spec == 1 ) {
        putchar(007);
        return;
    }
    if ( enter_errs == 1 ) {
        putchar(007);
        warning(w, "Please close error message window");
        return;
    }
    enter_spec = 1;

    dialog = XtVaCreatePopupShell ("dialog", xmDialogShellWidgetClass,
                                   XtParent(w),
                                   XmNtitle, "Prototype Specification",
                                   XmNdeleteResponse, XmDESTROY,
                                   NULL);

    action_items[1].data = (XtPointer)dialog; //Set cancel buttons
    client_data

    pane = XtVaCreateWidget("pane", xmPanedWindowWidgetClass, dialog,
                           XmNsashWidth, 1,
                           XmNsashHeight, 1,
                           NULL);

    rc = XtVaCreateWidget("control_area", xmRowColumnWidgetClass, pane,
                           NULL);
    string = XmStringCreateSimple("View or Edit Prototype Specification");
    XtVaCreateManagedWidget("label", xmLabelGadgetClass, rc,
                           XmNlabelString, string,
                           NULL);
    XmStringFree(string);

    description = graphic_list.cur_op_spec();

    int      n = 0;
    Arg      args[10];
    XtSetArg(args[n], XmNrows, 12); n++;
    XtSetArg(args[n], XmNcolumns, 70); n++;
    XtSetArg(args[n], XmNscrollVertical, true); n++;
    XtSetArg(args[n], XmNscrollHorizontal, true); n++;
    XtSetArg(args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;

```

```

XtSetArg(args[n], XmNeditable, true); n++;
XtSetArg(args[n], XmNcursorPositionVisible, true); n++;
XtSetArg(args[n], XmNwordWrap, true); n++;
XtSetArg(args[n], XmNvalue, description); n++;
text_w = XmCreateScrolledText(rc, "text-field", args, n);
XtManageChild(text_w);
//text_w = XtVaCreateManagedWidget("text-field",
xmTextFieldWidgetClass,
// rc, NULL);

// XtAddCallback(text_w, XmNmodifyVerifyCallback, validate_text,
NULL);
// Note: If you have problems with '}' symbols in the text,
uncomment
// the line above.

XtManageChild(rc);

//Set client data for the "OK" and "Cancel" buttons
action_items[0].data = (XtPointer)text_w;

action_a = CreateActionArea(pane, action_items,
XtNumber(action_items));

//XtAddCallback(text_w, XmNactivateCallback, activate_cb, action_a);

XtManageChild(pane);

/*
 * MY
 *
 * if (strcmp(XmTextGetString(text_w), "") == 0 ||
 *      strcmp(XmTextGetString(text_w), "\n") == 0 )
 *      XmTextInsert (text_w, 0L, "SPECIFICATION\nEND");
 *
 *
 */

free(description);
XtPopup(dialog, XtGrabNone);

}

// Callback function. Called when the radio buttons in the
// properties dialog box are pushed. Called twice: once to
// unselect old button, again to select the new one.

void radio_box_cb(Widget, XtPointer which,
                  XtPointer cbs) {
    XmToggleButtonCallbackStruct *state =
        (XmToggleButtonCallbackStruct *) cbs;

    if (state->set) {
        if ((int) which == 0)
            state_stream = false;
        else
            state_stream = true;
    }
}

```



```

void save_indicator_cb(Widget widget, XtPointer,
                      XtPointer cb_struct_ptr) {

    if (psdl_modified)
        handle_file_options(0);    // save

}

void error_indicator_cb(Widget widget, XtPointer client_data,
                      XtPointer call_data) {
/*
 * MY 7/22/97
 */

    if ((errors_present == NULL) || (!syntax_checked))
        handle_psdl_options(3);    // check syntax
    else {
        report_errors(errors_present, toplevel, next_action_ptr,
                      &return_sde_flag, &prev_status);
    }
/*
 * unmasked 8/6/97
 */

}

// If graph editor is invoked in viewer mode, this function
// handles ClientMessage events from the syntax-directed editor.
// Commented-out code handles data passed in a property, which
// this version of the editor doesn't take advantage of.
// Used during testing, and left in for future use, if necessary.

void event_handler(Widget widget, XtPointer,
                  XEvent* in_event, Boolean*) {

    char buffer[INPUT_LINE_SIZE];
    Display *display_ptr = XtDisplayOfObject(widget);
    Window window = DefaultRootWindow(display_ptr);
    // char **data;
    // int return_count;
    // XTextProperty text_prop_return;
    // Atom property_name;
    char message_in[30];

    strcpy(message_in, in_event->xclient.data.b);
    if (strcmp(message_in, "GEDATAIN") == 0) {

        graphic_list.build_from_sde(gdnode);          // @3

#ifdef GE_DEBUG
        // printf("graphic_list: after build\n");
        // graphic_list.summarize();
#endif

        // graphic_list.build_from_disk();    // @3
        graphic_list.draw();
    }
}

```

```

else if (strcmp(message_in, "PrintWindow") == 0)
{
    if (PrintCmd.op == Snd_to_Prt) {
        if ((PrintCmd.printer != NULL) && (*PrintCmd.printer != '\0')) {
            sprintf(buffer,
                "xwd -frame -id %d | xpr -gray 2 -device ps | lpr -P%s ",
                XtWindow(toplevel), PrintCmd.printer);
        }
        else {
            sprintf(buffer,
                "xwd -frame -id %d | xpr -gray 2 -device ps | lpr ",
                XtWindow(toplevel));
        }
        setcursor(toplevel, True, XC_watch);
        system(buffer);
        setcursor(toplevel, True, XC_left_ptr);
    }
    else {
        if ((PrintCmd.file != NULL) && (*PrintCmd.file != '\0')) {
            sprintf(buffer,
                "xwd -frame -id %d > %s ",
                XtWindow(toplevel), PrintCmd.file);
        }
        else {
            warning(drawing_a, "A file name must be supplied.");
        }
        setcursor(toplevel, True, XC_watch);
        system(buffer);
        setcursor(toplevel, True, XC_left_ptr);
    }
}
else {
#ifdef GE_DEBUG
    // cout << "Event " << message_in << endl;
#endif
}
}

```

```

void set_current_op() {
    char *cur_op_name;

    cur_op_name = graphic_list.current_op_name();
    if (cur_op_name != NULL)
        XtVaSetValues(current_op_name, XmNvalue, cur_op_name, NULL);
    free(cur_op_name);
}

```

```

void set_current_op_met() {
    char buffer[25] = "MET ";
    char *time;
    char *met = buffer;

    if (graphic_list.cur_op_spec_met() != UNDEFINED_TIME) {
        time = time_with_units(graphic_list.cur_op_spec_met(),
            graphic_list.cur_op_spec_met_unit());
        strncat(met, time, 20);
        XtVaSetValues(current_op_met, XmNvalue, met, NULL);
        free(time);
    }
}

```

```

    }
    else
        XtVaSetValues(current_op_met, XmNvalue, "", NULL);
}

void set_editor_title() {

    char    title_str[63] = "PSDL Editor: \0";
    char    *title_ptr = title_str;
    char    *name_ptr;

    name_ptr = graphic_list.root_op_name();
    if (name_ptr != NULL)
        strncat(title_ptr, name_ptr, 50);
    XtVaSetValues(toplevel, XmNtitle, title_ptr, NULL);
    free(name_ptr);
}

void init_motif() {

    // Simulated arguments
    // char*  args[] = {"edit_graph", "-geometry", "800x600", NULL};
    // int    signed_argc = 3;
    // char** argv = args;

    char*  args[] = {"edit_graph", "-geometry", "800x600", NULL};
    int    Global_argc = 3;
    char** Global_argv = args;

    char    title_str[63] = "PSDL Editor: \0";
    char    *title_ptr = title_str;
    XmString tmp;

    print_event = (XEvent *) malloc(sizeof(XEvent));    // @7

    toplevel = XtVaAppInitialize(&app, "edit_graph", options,
                                XtNumber(options), &Global_argc,
Global_argv,
                                NULL, NULL);

    display_ptr = XtDisplay(toplevel);
    XtGetApplicationResources(toplevel, (XtPointer) &Resrcs,
                                resources, XtNumber(resources),
                                NULL, 0);
    screen_ptr = XtScreen(toplevel);
    initialize_color_table(screen_ptr);
    root_window = RootWindowOfScreen(screen_ptr);
    gcv1.foreground = BlackPixelOfScreen(screen_ptr);
    gcv1.background = WhitePixelOfScreen(screen_ptr);
    gcv2.foreground = BlackPixelOfScreen(screen_ptr);
    gcv2.background = WhitePixelOfScreen(screen_ptr);
    gcv3.foreground = WhitePixelOfScreen(screen_ptr);
    gcv3.background = WhitePixelOfScreen(screen_ptr);
    gc_mask = GCForeground | GCBackground;
    std_graphics_context = XCreateGC(display_ptr,
                                root_window, gc_mask, &gcv1);
    dotted_context = XCreateGC(display_ptr,
                                root_window, gc_mask, &gcv2);
    erase_context = XCreateGC(display_ptr, root_window, gc_mask,
                                &gcv3);
}

```

```

XtSetLineAttributes(display_ptr, dotted_context, 1,
                    LineOnOffDash, CapButt, JoinMiter);
XtSetFunction(display_ptr, dotted_context, GXxor);

main_w = XtVaCreateManagedWidget("main_w", xmFormWidgetClass,
                                toplevel, NULL);
build_menu_bar(main_w, menubar);
XtManageChild(menubar);
rowcol =
    XtVaCreateManagedWidget("rowcol", xmRowColumnWidgetClass,
                            main_w,
                            XmNnumColumns, 1,
                            //      XmNorientation, XmHORIZONTAL,
                            NULL);

make_buttons(rowcol,
            op_button, term_button, stream_button, select_button,
            types_button, spec_button,
            timers_button, informal_button,
            op_button_pixmap, term_button_pixmap,
            stream_button_pixmap, select_button_pixmap,
            types_button_pixmap, spec_button_pixmap,
            informal_button_pixmap, timers_button_pixmap,
            display_ptr, screen_ptr);

XtAddCallback(op_button,      XmNactivateCallback, op_button_cb,
NULL);
XtAddCallback(term_button,    XmNactivateCallback, term_button_cb,
NULL);
XtAddCallback(stream_button,  XmNactivateCallback, stream_button_cb,
NULL);
XtAddCallback(select_button,  XmNactivateCallback, select_button_cb,
NULL);
XtAddCallback(types_button,   XmNactivateCallback, types_button_cb,
NULL);
XtAddCallback(spec_button,    XmNactivateCallback, spec_button_cb,
NULL);
XtAddCallback(timers_button,  XmNactivateCallback, timers_button_cb,
NULL);
XtAddCallback(informal_button, XmNactivateCallback,
informal_button_cb, NULL);

XtVaSetValues(toplevel, XmNtitle, title_ptr, NULL);

current_op_name =
    XtVaCreateManagedWidget("current_op_name", xmTextWidgetClass,
                            main_w,
                            XmNvalue, "",
                            XmNshadowThickness, 1,
                            NULL);

current_op_met =
    XtVaCreateManagedWidget("current_op_met", xmTextWidgetClass,
                            main_w,
                            XmNvalue, "",
                            XmNwidth, 150,
                            XmNshadowThickness, 1,
                            NULL);

scrolled_win =
    XtVaCreateManagedWidget("scrolled_win",

```

```

        xmScrolledWindowWidgetClass,
        main_w,
        // XmNwidth, 1200,
        // XmNheight, 750,
        XmNscrollingPolicy, XmAUTOMATIC,
        XmNscrollBarDisplayPolicy, XmAS_NEEDED,
        NULL);
actions.string = "draw";
actions.proc = draw;
XtAppAddActions(app, &actions, 1);
status_indicator =
    XtVaCreateManagedWidget("status_indicator", xmTextWidgetClass,
        main_w,
        XmNheight, 31,
        XmNvalue, "",
        NULL);

save_indicator =
    XtVaCreateManagedWidget("save_indicator",
        xmDrawnButtonWidgetClass,
        main_w,
        XmNrecomputeSize, false,
        XmNpushButtonEnabled, false,
        XmNshadowType, XmSHADOW_IN,
        XmNwidth, 120,
        XmNheight, 31,
        XmNmarginBottom, 13,
        XmNlabelType, XmSTRING,
        NULL);
XtAddCallback(save_indicator, XmNactivateCallback,
save_indicator_cb, NULL);

error_indicator =
    XtVaCreateManagedWidget("error_indicator",
        xmDrawnButtonWidgetClass,
        main_w,
        XmNrecomputeSize, false,
        XmNpushButtonEnabled, false,
        XmNshadowType, XmSHADOW_IN,
        XmNwidth, 120,
        XmNheight, 31,
        XmNmarginBottom, 13,
        XmNlabelType, XmSTRING,
        NULL);
XtAddCallback(error_indicator, XmNactivateCallback,
error_indicator_cb, NULL);

drawing_a =
    XtVaCreateManagedWidget("drawing_a",
        xmDrawingAreaWidgetClass, scrolled_win,
        XmNunitType, Xm1000TH_INCHES,
        XmNwidth, 11000,
        XmNheight, 8500,
        XmNresizePolicy, XmNONE,
        NULL);
XtAddCallback(drawing_a, XmNexposeCallback, redraw, NULL);

XtVaSetValues(drawing_a, XmNunitType, XmPIXELS, NULL);
XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);
XtVaGetValues(drawing_a, XmNwidth, &width, XmNheight, &height,
    NULL);

```

```

drawing_area_pixmap = XCreatePixmap(display_ptr,
                                     root_window, width, height,
                                     DefaultDepthOfScreen(screen_ptr));
XFillRectangle(display_ptr, drawing_area_pixmap,
               erase_context, 0, 0, width, height);

XtVaSetValues(drawing_a, XmNtranslations,
              XtParseTranslationTable(translations), NULL);

XtVaSetValues(rowcol,
              XmNtopAttachment,    XmATTACH_FORM,
              XmNrightAttachment,  XmATTACH_NONE,
              XmNleftAttachment,  XmATTACH_FORM,
              XmNbottomAttachment, XmATTACH_WIDGET,
              XmNbottomWidget,    save_indicator,
              NULL);

XtVaSetValues(menubar,
              XmNtopAttachment,    XmATTACH_FORM,
              XmNrightAttachment,  XmATTACH_FORM,
              XmNleftAttachment,   XmATTACH_WIDGET,
              XmNleftWidget,       rowcol,
              XmNbottomAttachment, XmATTACH_NONE,
              NULL);

XtVaSetValues(current_op_name,
              XmNtopAttachment,    XmATTACH_WIDGET,
              XmNtopWidget,        menubar,
              XmNrightAttachment,  XmATTACH_WIDGET,
              XmNrightWidget,      current_op_met,
              XmNleftAttachment,   XmATTACH_WIDGET,
              XmNleftWidget,       rowcol,
              XmNbottomAttachment, XmATTACH_NONE,
              NULL);

XtVaSetValues(current_op_met,
              XmNtopAttachment,    XmATTACH_WIDGET,
              XmNtopWidget,        menubar,
              XmNrightAttachment,  XmATTACH_FORM,
              XmNleftAttachment,   XmATTACH_NONE,
              XmNbottomAttachment, XmATTACH_NONE,
              NULL);

XtVaSetValues(scrolled_win,
              XmNtopAttachment,    XmATTACH_WIDGET,
              XmNtopWidget,        current_op_name,
              XmNrightAttachment,  XmATTACH_FORM,
              XmNleftAttachment,   XmATTACH_WIDGET,
              XmNleftWidget,       rowcol,
              XmNbottomAttachment, XmATTACH_WIDGET,
              XmNbottomWidget,     status_indicator,
              NULL);

XtVaSetValues(save_indicator,
              XmNtopAttachment,    XmATTACH_NONE,
              XmNrightAttachment,  XmATTACH_NONE,
              XmNleftAttachment,   XmATTACH_FORM,
              XmNbottomAttachment, XmATTACH_FORM,
              NULL);

XtVaSetValues(error_indicator,

```

```

        XmNtopAttachment,      XmATTACH_NONE,
        XmNrightAttachment,    XmATTACH_NONE,
        XmNleftAttachment,     XmATTACH_WIDGET,
        XmNleftWidget,         save_indicator,
        XmNbottomAttachment,    XmATTACH_FORM,
        NULL);

XtVaSetValues(status_indicator,
        XmNtopAttachment,      XmATTACH_NONE,
        XmNrightAttachment,    XmATTACH_FORM,
        XmNleftAttachment,     XmATTACH_WIDGET,
        XmNleftWidget,         error_indicator,
        XmNbottomAttachment,    XmATTACH_FORM,
        NULL);

XtRealizeWidget(toplevel);
draw_window = XtWindow(drawing_a);

graphic_list.set_draw_environ(display_ptr,
        std_graphics_context,
        erase_context, dotted_context,
        draw_window,
        &drawing_area_pixmap,
        color_table,
        width, height);
graphic_list.set_error_tgt(drawing_a);

set_current_op();
set_current_op_met();

XmProcessTraversal(drawing_a, XmTRAVERSE_CURRENT);

toplevel_window = XtWindowOfObject(toplevel);
Atom display_id_atom = XInternAtom(display_ptr, "WINDOW_ID",
        False);
XChangeProperty(display_ptr, root_window, display_id_atom,
        XA_WINDOW, 32, PropModeReplace,
        (unsigned char *) &toplevel_window, 1);
XtAddEventHandler(toplevel, NoEventMask, true, event_handler,
        NULL);

motif_initialized = true;
}

// translations provides the mappings for the keyboard
// mapping table that allow the drawing canvas to capture
// mouse and keyboard events.

// The primary function, edit_graph. Modified from original main() by
// Doug Lange 9/9/96

/*****
* this method is added to support the edit graph and sde change over.
* now the edit graph module is not a standalone but a method called
* from the sde
*****/

//extern "C" {

```

```

/*
 * modified 7/12/97
 * int edit_graph(...) -> void edit_graph(...)
 */

void edit_graph(GRAPH_DESC current_graph, ACTION next_action,
                ERROR_MSGS sde_error_msgs) {    //@2

    XEvent event; // added for custom main loop

    int reply;
    Quest_Script delete_script =
        {"", "Deleted operators will be purged?", "Ok", "No", "Cancel",
        BTN1};

    next_action_ptr = next_action;
    errors_present = sde_error_msgs;
    return_sde_flag = false;

    gdnnode = current_graph;

    // motif_initialized assumed to be false at start of procedure

    if (motif_initialized) {
        XFillRectangle(display_ptr, drawing_area_pixmap,
                        erase_context, 0, 0, width, height);
        XFillRectangle(display_ptr, draw_window,
                        erase_context, 0, 0, width, height);
        if (gdnnode) {
            graphic_list.build_from_sde(gdnnode);
        }

        if (save_performed)
            save_state(NOT_MODIFIED);
        if (prev_status) {
            update_status(prev_status, false);
            free(prev_status);
            prev_status = NULL;
        }
        graphic_list.draw();
        setcursor(toplevel, True, XC_left_ptr);
    }
    else {
        init_motif();
        prev_status = NULL;
        save_state(NOT_MODIFIED);
        save_performed = false;
        default_color = WHITE;
        default_font = COURIERBOLD12;
        graphic_list.set_default_font(default_font);
        if (gdnnode) {
            graphic_list.build_from_sde(gdnnode);
        }

        graphic_list.draw();

        select_state(SELECT_TOOL);

        // Initialize printer command
        PrintCmd.op = Snd_to_Prt;
        PrintCmd.printer = dup_str("");
    }
}

```



```

PrintCmd.file    = dup_str("");
PrintCmd.answer  = 0;

// and event
print_event->type = ClientMessage;
print_event->xclient.window = toplevel_window;
print_event->xclient.format = 8;
strcpy(print_event->xclient.data.b, "PrintWindow");
}
set_editor_title();
set_current_op();
set_current_op_met();

syntax_checked = true;    // syntax is checked on each entry to editor
error_label();

if (graphic_list.cur_op_is_terminator())
    XtVaSetValues(op_button, XmNsensitive, False, NULL);
else
    XtVaSetValues(op_button, XmNsensitive, True, NULL);

// MY // printf("\n"); //flushes the event queue
XFlush(display_ptr);

#ifdef GE_DEBUG
    cout << "Starting Motif event loop" << endl;
#endif

selected_object_ptr = NULL;

// Custom main loop to check for return to sde
do {
    XtAppNextEvent(app, &event);
    XtDispatchEvent(&event);

    if (return_sde_flag) {
        if (graphic_list.has_deleted()) {
            reply = AskUser(app, drawing_a, delete_script);
            if (reply != YES)
                return_sde_flag = false;
        }
    }

} while (return_sde_flag == false);

if ((next_action->option != REVERT) &&
    (next_action->option != ABANDON))
    graphic_list.write_to_sde(gdnode);

if ((next_action_ptr->option == SAVE_TO_DISK) ||
    (next_action_ptr->option == REVERT)) // not really saved, but not
    save_performed = true;              // modified
else
    save_performed = false;              // assume need to save for
abandon

// If we are not coming back, kill the window
if (!next_action_ptr->reinvoke) {
    XtUnrealizeWidget(toplevel);
    XFlush(display_ptr);
}

```

```

    else { // otherwise, will be returning, erase window and exit
        setcursor(toplevel,True,XC_watch);
    }
    return;
}

//} // extern "C"

/*****
*          --- end of graph_editor.C
*****/

```

LIST OF REFERENCES

- [BATANI86] C. Batani, E. Nardelli, R. Tamassia, "A Layout Algorithm for Data Flow Diagrams", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 4, pp. 538-546, April 1986.
- [BECKER95] R. Becker, S. Eick, A. Wilks, "Visualizing Network Data", *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 1, pp. 16-28, March 1995.
- [CUMMINGS90] Cummings, Mary Ann, "The Development of User Interface Tools for the Computer-Aided Prototyping System", *M.S. Thesis*, Naval Postgraduate School, Monterey, California, December 1990.
- [HIMSOLT96] M. Himsolt, "GML: Graph Modelling Language", *Draft Version*, December 1996.
- [HIMSOLT97] M. Himsolt, World Wide Web Home Page, <http://vogelweide.fmi.uni-passau.de/~himsolt/graphdrawing.html>, August 1997.
- [KARP94] P. Karp, J. Lowrance, T. Strat, D. Wilkins, "The Grasper-CL Graph Management System", *Lisp and Symbolic Computation*, no. 7, pp. 245-273, 1994.
- [LUQI88-1] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System", *IEEE Software*, March 1988.
- [LUQI88-2] Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering*, October 1988.
- [LUQI89] Luqi, "Handling Timing Constraints in Rapid Prototyping", *Proceedings of the Twenty-Second Annual Hawaii Conference on Systems Science*, January 1989.
- [LUQI90] Luqi, "A Graph Model for Software Evolution", *IEEE Transactions on Software Engineering*, August 1990.
- [LUQI92] Luqi, "Computer-Aided Prototyping for a Command-and-Control System Using CAPS", *IEEE Software*, January 1992.

- [TAMASSIA94] R. Tamassia, P. Eades, "Algorithms for Drawing Graphs: an Annotated Bibliography", *Technical Report No. CS-89-09*, June 1994.
- [TAMASSIA97] R. Tamassia, World Wide Web Home Page, <http://www.cs.brown.edu/people/rt>, August 1997.
- [TOLLIS97] I. Tollis, World Wide Web Home Page, <http://www.utdallas.edu/dept/cs/tollis.html>, August 1997.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
8725 John J. Kingman Road, Suite 0944
Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library.....2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943

3. Center for Naval Analysis.....1
4401 Ford Ave.
Alexandria, VA 22302

4. Dr. Ted Lewis, Chairman, Code CS/Lt.....1
Computer Science Dept.
Naval Postgraduate School
Monterey, CA 93943

5. Chief of Naval Research.....1
800 North Quincy St.
Arlington, VA 22217

6. Dr. Luqi, Code CS/Lq.....1
Computer Science Dept.
Naval Postgraduate School
Monterey, CA 93943

7. Dr. Marvin Langston.....1
1225 Jefferson Davis Highway
Crystal Gateway 2 / Suite 1500
Arlington, VA 22202-4311

8. David Hislop.....1
U.S. Army Research Office
PO Box 12211
Research Triangle Park, NC 27709-2211

9. Capt. Talbot Manvel.....1
 Naval Sea System Command
 2531 Jefferson Davis Hwy.
 Attn: TMS 378 Capt. Manvel
 Arlington, VA 22240-5150

10. CDR Michael McMahon.....1
 Naval Sea System Command
 2531 Jefferson Davis Hwy.
 Arlington, VA 22242-5160

11. Dr. Elizabeth Wald.....1
 Office of Naval Research
 800 N. Quincy St.
 ONR CODE 311
 Arlington, VA 22217-5660

12. Dr. Ralph Wachter.....1
 Office of Naval Research
 800 N. Quincy St.
 CODE 311
 Arlington, VA 22217-5660

13. Army Research Lab.....1
 115 O'Keefe Building
 Attn: Mark Kendall
 Atlanta, GA 30332-0862

14. National Science Foundation.....1
 Attn: Bruce Barnes
 Div. Computer & Computation Research
 1800 G St. NW
 Washington, DC 20550

15. National Science Foundation.....1
 Attn: Bill Agresty
 4201 Wilson Blvd.
 Arlington, VA 22230

16. Hon. John W. Douglass.....1
Assistant Secretary of the Navy
(Research, Development and Aquisition)
Room E741
1000 Navy Pentagon
Washingotn, DC 20350-1000
17. Technical Library Branch.....1
Naval Command, Control, and Ocean Surveillance Center
RDT&E Division, Code D0274
San Diego, CA 92152-5001
18. Head, Command and Control Department.....1
Naval Command, Control and Ocean Surveillance Center
RDT&E Division, Code D40
San Diego, CA 92152-5001
19. Head, Simulation and Human Systems Technology Division.....1
Naval Command, Control and Ocean Surveillance Center
RDT&E Division, Code D44
San Diego, CA 92152-5001