André Emanuel Bernardo Almeida

Towards Automata Diagram Drawings

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto

2010

André Emanuel Bernardo Almeida

Towards Automata Diagram Drawings

Tese submetida à Faculdade de Ciências da
Universidade do Porto para obtenção do grau de Mestre
em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2010

To my parents.

Acknowledgments

I would like to thank Nelma Moreira and Rogério Reis, for all their guidance and commitment, that helped me through my academic formation and specially in my Master's Degree.

I would also like to thank José Daniel da Silva Alves for the critic spirit that he had through all time that we worked together.

Contrarily to what most people think, my parents, Manuel Pinto Bernardo and Maria de Almeida Pinto Cardoso, helped me a lot by supporting my decisions and encouraging me to proceed my studies.

Abstract

Graphs are often used to show information in many applications. They are capable of displaying the information in a more pleasant way then other formats and can simplify its interpretation. The finite automata diagrams are a good example of this. Finite automata are drawn using a set of conventions that intend to improve their readability. Currently there are no good tools available for automatic drawing of finite automata, that respect these conventions. GUItar is a graphical environment for the visualization, editing, and interaction of diagrams, that specially focuses in finite automata type of diagrams. The application incorporates mechanisms to facilitate the edition of these diagrams. It also provides two style managers that allow the creation of rich node and arc styles to be used in the diagrams drawings. These style managers allow the system to cope with complex styles, broaden the application scope to graphical representations of other computational models like transducers or Turing machines. GUItar also has a foreign function call (FFC) mechanism for the easy integration of external modules and libraries like automata symbolic manipulators or graph drawing libraries. FAgoo is a Python module that seeks to provide a set of graph drawing algorithms for finite automata diagrams. Currently FAgoo implements some important graph algorithms, such as planarity testing and planar embedding. Both GUItar and FAgoo are on going projects licensed under GPL. The work of this thesis was the implementation of a graphical environment for finite automata diagrams (GUItar) and a first version of a graph drawing library for finite automata diagrams (FAgoo).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Information is often represented by graphs in many applications because they can be displayed in a relatively small space. A good graph drawing can help a lot the interpretation of the information associated with it. If the graph is small, then it is easy to draw it, but usually graphs are very large and complex which hardens the task of drawing them. A good example of this are finite automata. Finite automata are usually drawn as diagrams to simplify its interpretation. Usually is easier to understand the behaviour of a finite automata when it is drawn as a diagram. These diagrams are drawn following a set of conventions, such as: initial states are positioned to the left, final states tend to be pushed to the right, and its readability flows from the left to the right. These conventions intend to improve the readability and understanding of these diagrams. If the finite automata is small, then it is easy to draw it and preserve these conventions, but finite automata that actually have a practical application and are used in the industry, are very large and complex, which difficult its drawing task. Graph drawing is an area with many years of research and development, and it is very well documented. There are many applications and libraries for graph drawing available, although most of them do not fit the finite automata drawing conventions, and the ones that are specifically designed to manipulate finite automata either present very poor graph drawing algorithms or are outdated.

GUItar [FAd10b] is a graphical environment tool for finite automata visualization and editing. The Figure 1.1 shows the interface of this application, with a drawing of a finite automaton. GUItar incorporates tools that assist the user through the diagram drawing and visualization processes, such as the automatic positioning of the nodes into a grid to avoid overlaps, the automatic adjustment of the arcs' control points and the automatic organization of the arcs so that arcs from the left to the right are positioned above the ones from the right to left. GUItar also provides powerful styling tools that not only allow the editing of node and arc styles but also allow the creation of new node structures. Furthermore, it implements a foreign function call (FFC) mechanism which is used to access external modules or libraries as FAdo and FAgoo [FAd10b]. FAdo is a tool for symbolic manipulation of formal languages, specially finite automata, that can be incorporated with GUItar. Since most FAdo

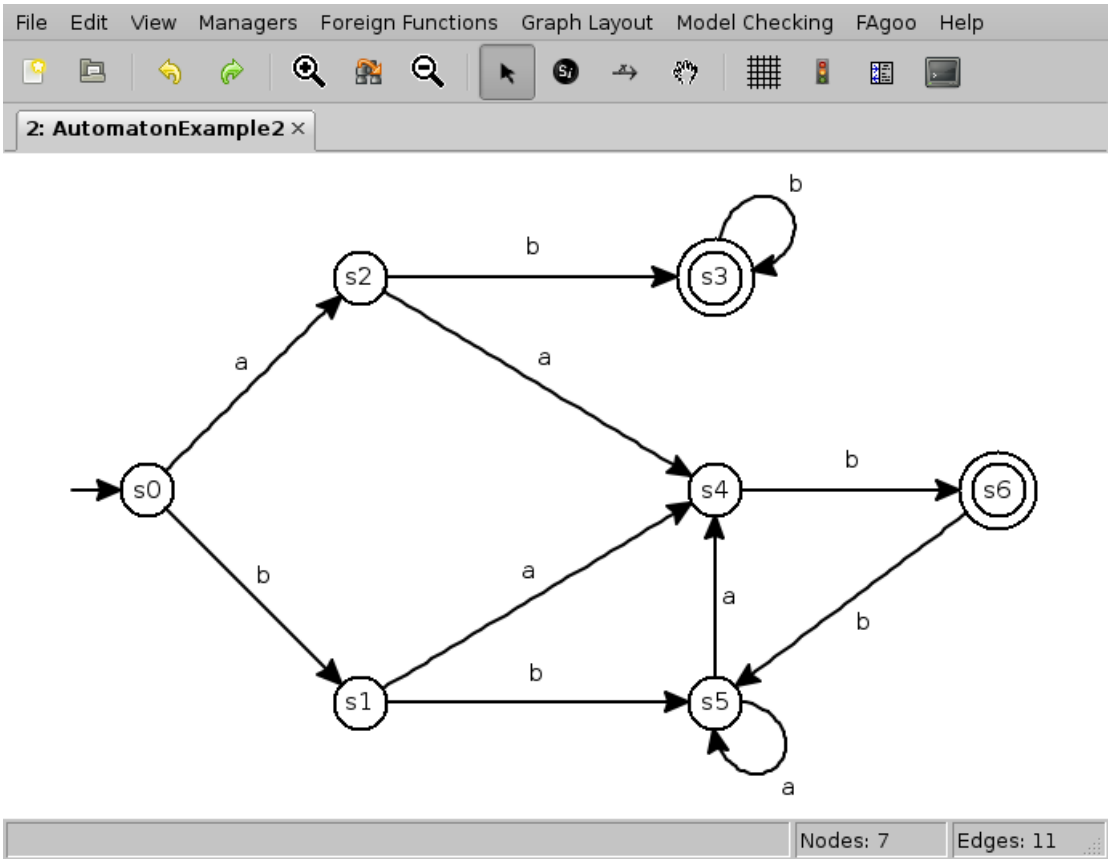


Figure 1.1: An automaton drawing created using GUItar.

manipulations result in finite automata with no embedding, the graph drawing library, FAgoo, is being developed. FAgoo specially focuses in finite automata type of diagrams, which require additional aesthetic and graphical constraints over other types of graphs. Already existing graph drawing algorithms must be adapted in order to fit the finite automata conventions. FAgoo is a Python module written in C, which provides a good performance and at the same time a high-level interface.

The project for this thesis intended to implement a graphical interface for finite automata visualization and editing, as well as a graph drawing library for finite automata. A working prototype of a graphical interface, GUItar, is presented in this thesis. It is also presented FAgoo, a first version of what intends to be graph drawing library specialized in finite automata drawings.

Chapter 2 gives some basic definitions in graph and finite automata theory. It also presents some graph drawing applications and libraries, and explains why they are not suitable for finite automata drawings. Chapter 3 presents GUItar, the implemented graphical interface for finite automata visualization and editing, and describes some of its features. FAgoo and some of the algorithms implemented in it are presented and described in Chapter 4. Finally Chapter 5 concludes this thesis and suggests some future work.

Chapter 2

Graphs and Automata Drawing Applications

2.1 Graphs and Drawings

It is usual in computer science to represent relational structures as *graphs*. The entities are represented by *vertices* and their relationships by *edges*. For instance in programming languages documentation, graphs are used to illustrate class dependencies. Each class is represented by a vertex, and every time a class b extends a class a there is an edge from a to b. The graph in Figure 2.1 illustrates some class dependencies in GUItar's canvas. Graphs are useful because they are capable of precisely display information and at the same time provide an easy way to read and interpret it. In order to achieve this, graphs must be drawn with a good *layout* that can be easy to follow and understand. Graphs with poor layouts are confusing, thus difficult to understand. The same graph can have some layouts that favor its readability and others that just difficult its readability. Figure 2.2a and Figure 2.2b represent the class dependencies in GUItar, where vertices represent the classes and edges represent the instantiation of one class in another, i.e., there is an edge *(u,v)* every time the class u contains a instance of the class v. Both Figure 2.2a and Figure 2.2b represent

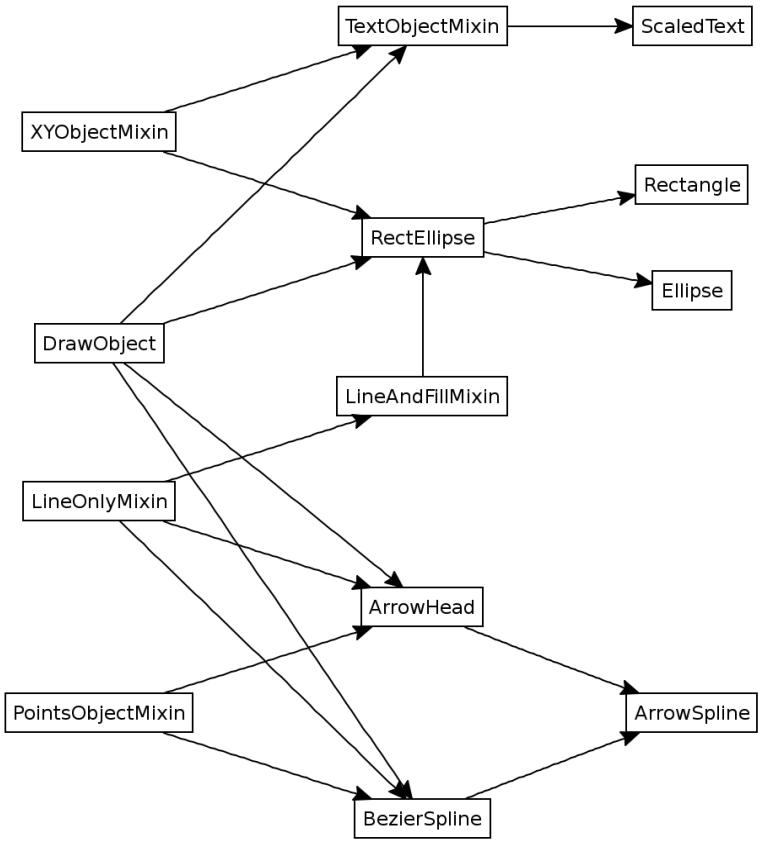
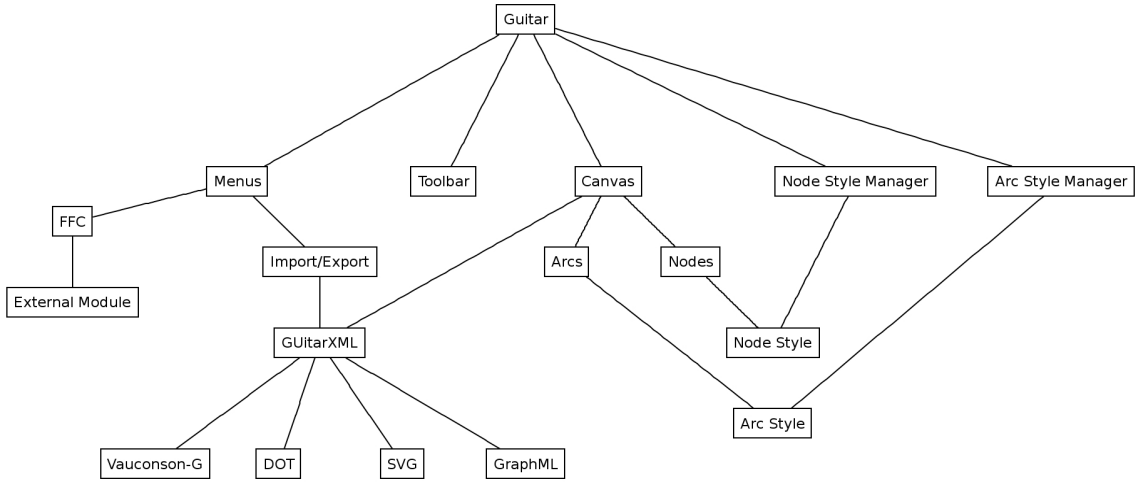


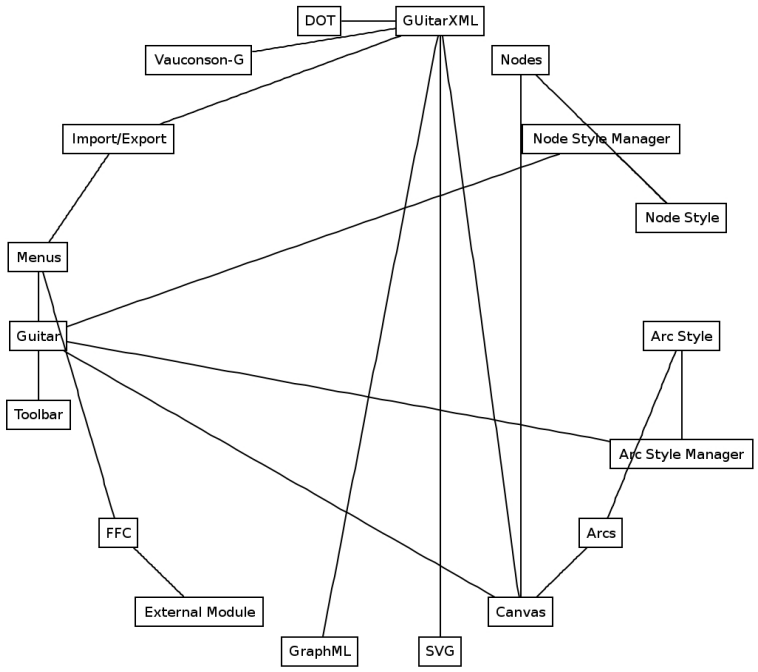
Figure 2.1: Some class dependencies in GUItar's FloatCanvas.

the same graph but with different layouts. While Figure 2.2a is easy to read and to understand, Figure 2.2b is confusing and difficult to follow.

A graph G is a tuple *(V,E)*, being V the set of *vertices* and E the set *edges* where each edge is an unordered pair *(u,v)* of vertices. In some contexts vertices are called *nodes* and edges are called *links*, *connections* or *arcs*. Two vertices u and v are *adjacent* if there is an edge *e=(u,v)* and e is called *incident* to u and v. The vertices u and v are also called the *end-points* of e. The vertex *neighbors* are its adjacent vertices. The *degree* ($deg(u)$) of a vertex u is the number of its incident edges. An edge *(u,v)* with $u = v$ is a $self\ loop$. An edge is a *multiple edge* if it occurs more than once. A graph with no self loops and no multiple edges is called a *simple graph*. Usually in generic graph algorithms, when nothing else is specified, simple graphs are assumed.



(a) A GUItar components overview with a layout without edge crossing.



(b) A GUItar components overview with a layout with edge crossing.

Figure 2.2: Two different layouts for the same graph.

A *directed graph*, also called *digraph*, is a graph where E is a set of ordered pairs of vertices (*directed edges*). A directed edge *(u,v)* is an *outgoing edge* of u and an *incoming edge* of v. A vertex is called a *source* if it only has incoming edges and a *sink* if it only has outgoing edges. In digraphs, a vertex degree can be distinguish

between *indegree* and *outdegree* which is the number of incoming and outgoing edges, respectively.

A *path* is a sequence $(v_1, v_2, ..., v_j)$ of distinct vertices, being each pair (v_i, v_{i+1}) of the sequence, with $1 \leq i \leq j$, an edge of the graph. If $v_1 = v_j$ then the path is called a *cycle*. A graph is *acyclic* if it has no cycle path, otherwise is a *cycle graph*.

A graph *G' = (V',E')* is a *subgraph* of *G = (V,E)* if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$.

There are two conventional ways of describing a graph, with an *adjacency matrix* or with an *adjacency list*. An adjacency matrix M of a graph *G = (V,E)* with $|V| = n$ and $|E| = m$ uses a $n \times n$ matrix such that, for every $u,v \in V$, $M_{uv} = 1$ if there is an edge *(u,v)*, $M_{uv} = 0$ otherwise. The other way of describing a graph is by using an adjacency list L_u for each vertex u, consisting on its adjacent vertices. For digraphs the list L_u only contains the adjacent vertices of the outgoing edges. Adjacency matrices can be easier to implement and access than adjacency lists, although an adjacency matrix uses space of size n^2 and to list the neighbors of a vertex u it takes an execution time bounded by n, while the adjacency list uses at most space of size $n+m$ and to list the neighbors of a vertex u it takes an execution time bounded by *deg(u)*.

A *drawing* is a family of mappings:

$$\Gamma : V \to \mathbb{R}^2, \Gamma_{u,v} : [0,1] \to \mathbb{R}^2,$$

where each $\Gamma_{u,v}$ must be injective and continuous, and such that $\Gamma_{u,v}(0) = \Gamma(u)$ and $\Gamma_{u,v}(1) = \Gamma(v)$.

Each vertex u is mapped to a distinct point Γ*(u)* in the plane and each edge is mapped to a simple curve Γ*(u,v)* with Γ*(u)* and Γ*(v)* as its endpoints. In digraphs the edges are normally drawn using arrows. It is also usual to refer to an edge drawing Γ*(u,v)* using the edge terminology *(u,v)*. A graph and its graph drawing are two different things, in fact a graph can have an infinite number of drawings.

A graph can be either *planar* or *non−planar*. To be planar it must be possible to draw it in a plane without edge crossings, i.e., if there are no pair of edges that intersect each other. The theory of planar graphs is well developed and documented. There are several algorithms to draw planar graphs described in literature. Edge crossing reduces the graph drawing readability making it confusing and difficult to follow. The non-planar graph drawing in Figure 2.2b is much more difficult to read than the one in Figure 2.2a. The Euler's formula implies that a simple planar graph with n vertices has at most $3n$ - 6 edges, which means that planar graphs can be drawn in a more scattered way.

A planar drawing induces an ordering on the neighbors of each vertex of the graph. The clockwise sequence of incident edges around each vertex is called an *embedding*. Two planar graph drawings of the same graph are *equivalent* when they have the same embedding. An *embedded graph* is a graph with an embedding associated. The graph drawings in Figure 2.3a, Figure 2.3b and Figure 2.3c are all equivalent (have the same embedding), and the one in Figure 2.3d is not (has a different embedding).

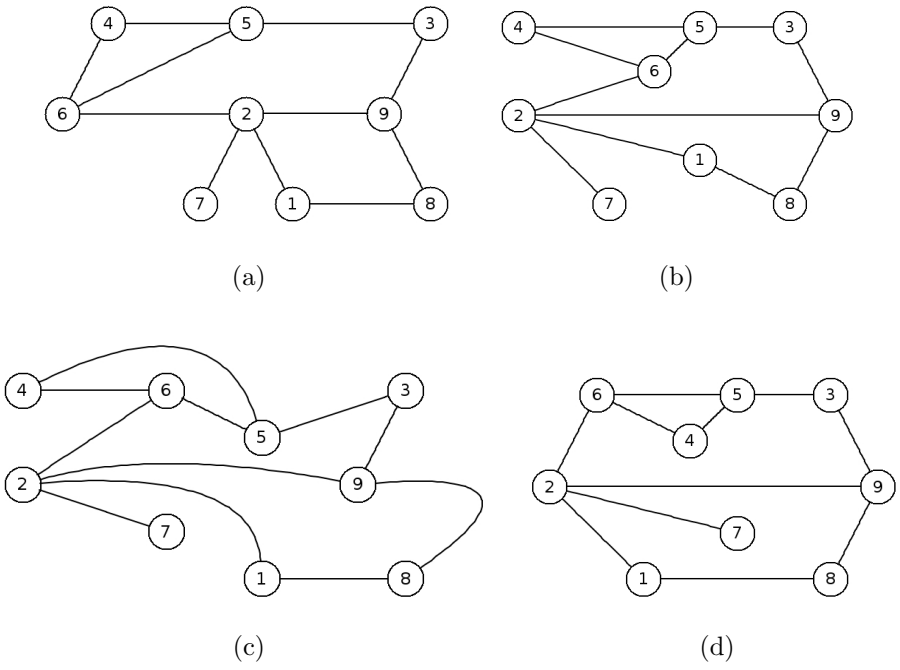


Figure 2.3: Four planar layouts for the same graph.

A planar drawing partitions the plane in connected regions which are called *faces*. Each face has associated a set of vertices and edges. The unbounded face is generally called the *external face*. The *dual graph* of a graph drawing has a vertex for each face and an edge *(u,v)* for each edge shared by the faces u and v. Notice that this graph may have self loops and multiple edges. An example of a dual graph is shown in Figure 2.4.

A graph is *connected* if for any pair of vertices u and v there is a path from u to v. A graph maximal connected subgraph is called a *connected component*. A *cutvertex* is a vertex that if removed, disconnects the graph. A *tree* is a connected graph without cycles (acyclic). A graph whose all connected components are acyclic (are trees) is called a *forest*. A graph is *biconnected* if it is connected and has no cutvertices. The maximal biconnected subgraph of a graph is called *block* or *biconnected component*. The *BC-tree* T_{bc} [Har69, 36] of a graph G is a graph where each block is represented by a *B-vertex* and each cutvertex by a *C-vertex*. There is an edge *(u,v)*, for each cutvertex represented by a C-vertex v, that belongs to a block represented by a B-vertex u. The BC-tree is sometimes denoted as the *block tree*, *bc(G)* or *2-block tree*. Biconnectivity notions and a BC-tree example are illustrated in Figure 2.5.

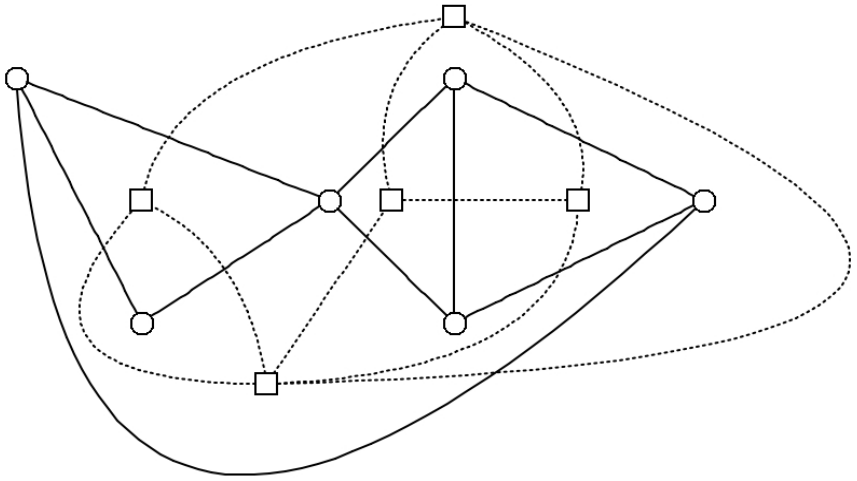
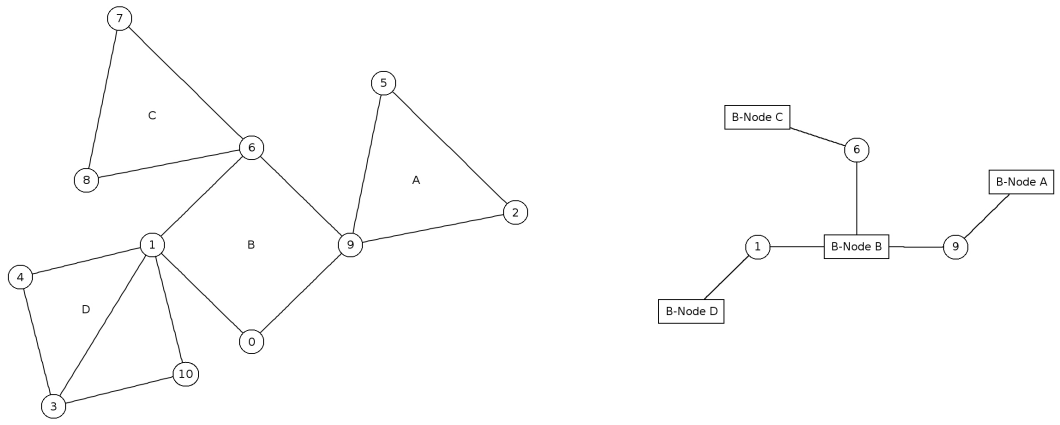


Figure 2.4: Dual graph example(represented by the squares and dashed lines).



(a) (b)

Figure 2.5: An example graph and its BC-tree(the letter represent the biconnected components).

A pair of vertices *(u,v)* in a biconnected graph G is a *separating pair* if their removal makes G disconnected. A biconnected graph with no separating pairs is called *triconnected*. Triconnected graphs are important in graph drawing. They can have interesting drawing properties. For example, a triconnected planar graph has only an unique embedding, up to a reversal of the circular ordering of the neighbors of each vertex.

Usually planar graph drawing algorithms use graph connectivity properties. For example a graph is planar if and only if all its biconnected components are. It is usual in graph drawing algorithms to assume that the input graph is biconnected. This is done because biconnected graphs have good properties that can simplify the algorithm and the biconnected components of a graph are relativily easy to compute.

2.2 Finite Automata

Automata are state based models, i.e., they model problems that can be represented by a finite set of states and transitions between them. A finite automaton usually has

at least one *initial state* and a set of acceptance states called *final states*. An input sequence is accepted if the resulting state is a *final state*. There are several applications of automata such as the designing and checking of digital circuits, parsers, lexical analyzers, software verification with model checking and definition of protocols. An example of an automaton that checks if the substring *FA* occurs in a string is illustrated in the Figure 2.6. The arrow pointing to the state *s0* indicates that it is the initial state. The automaton remains in state *s0* until the letter F occurs, in which case it moves to state *s1*. In state *s1* the automaton moves to *s2* if the input letter is an A, otherwise it returns to *s0*. Once in state *s2* the automaton remains in it disregarding the input letter. The two concentric ellipses used in state *s2* indicate that this state is final. If the automaton is in state *s2* after its execution then the substring *FA* occurs in the input string otherwise *FA* does not occur.

In the study of regular languages finite automata are used as its computational model. A finite automaton is deterministic if for each state there is not more than one transition with the same input, otherwise it is non-deterministic. The terminology DFA and NFA is usually used for deterministic and non-deterministic finite automata, respectively.

A NFA A consists in five elements and is usually represented as a tuple as it follows:

$$A = (Q , \Sigma , \delta , Q_0 , F)$$

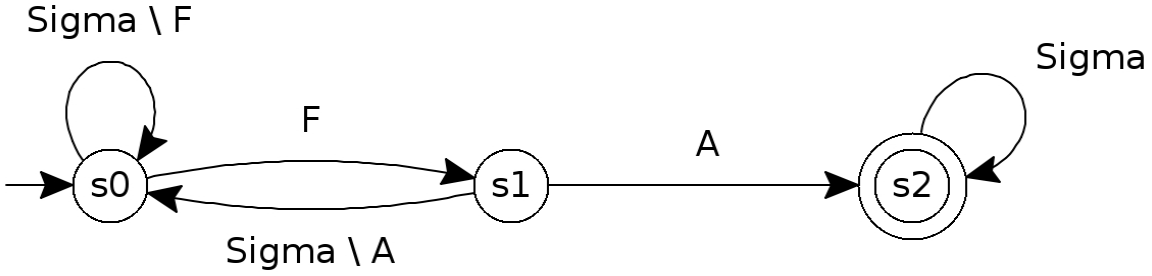


Figure 2.6: Automaton that accept strings containing the *FA* substring.

where Q is a finite set of *states*, Σ is a finite set of *input symbols*, δ is a *transition function* $Q \times \Sigma \rightarrow 2^Q$, Q_0 is a non-empty set of *initial states* with $Q_0 \subseteq Q$ and F is a set of *final states* with $F \subseteq Q$. The specification of the automaton in Figure 2.6 is

$$A = (\ \{s0,s1,s2\}\ ,\ \{A,B,C,...,F,...,Z\}\ ,\ \delta\ ,\ \{s0\}\ ,\ \{s2\}\ )$$

with the transition function

$$(s0,\Sigma\backslash F) \rightarrow s0 \quad (s0,F) \rightarrow s1 \quad (s1,\Sigma\backslash A) \rightarrow s0$$
$$(s1,A) \rightarrow s2 \quad (s2,\Sigma) \rightarrow s2.$$

Finite automata are often represented as diagrams like the one in Figure 2.6. Given the structure of a finite automata it is clear to consider its underline graph. Each state is represented by a labeled node. Usually all the nodes have the same shape (circle or ellipse) with the exception of the initial and final states. Normally the initial states have an arrow pointing to its circle (or ellipse) and final states are drawn as two concentric circle (or ellipse). The node's label is placed inside of it. These nodes are usually called states. For each transition $(q,a) \rightarrow q'$ there is a directed arc from the state q to the state q' labeled with the input symbol a. It is acceptable to have multiple arcs for the multiple transitions between two states, although these are normally represented by a single arc with their different input symbols concatenated by commas. The edges usually have their labels on their left side and edges from the left to the right are placed above the one from the right to the left. These diagrams are a special type of graph. They are labeled digraphs which can have self loops and are usually drawn according to some conventions that intend to improve its readability. The initial states are placed on the left so that the diagram readability flow from the left to the right, pushing the final states to the right. When the diagram is acyclic this is not difficult to achieve. When the diagram is cyclic and non-planar it is not so easy to keep the left to right readability.

2.3 Graph Drawing Libraries

There are several graph drawing libraries available with many layout algorithms for generic and specific types of graphs. Most of these libraries focus in a specific type of graphs in order to achieve better drawings. Restricting the type of graphs that an algorithm have to deal with, results in having graphs with particular properties which usually facilitates their drawings. Some of the most significant graph drawing libraries and applications are presented in this section.

2.3.1 aiSee

The application aiSee [aiS10] is currently used by several areas for graph visualization. The application is non-free and closed-source, although it can be used for non-commercial purposes with some restrictions, and a C library version is available for some costumers if requested. It provides a set of features that help the user to visualize large graphs. Graphs up to 1,000,000 nodes and 1,500,000 edges can be easily handle. The drawings can be exported to various formats such as PostScript (PS), Scalable Vector Graphics (SVG), and PNG. This software is used for many purposes such as software development, hardware design, web development and others. aiSee is currently used to draw many type of graphs, in particular:

- Trees;
- Organization charts;
- Control flow graphs;
- Circuit diagrams;
- Entity relationship diagrams;
- P2P networks;

- Network traffic graphs;
- Parse trees;
- Algorithm animation;
- Finite state diagrams.

Figure 2.7 shows a finite automaton example drawn using aiSee with the depth first search layout algorithm. Other available layout algorithms did not present better results. There are currently fifteen layout algorithms available which are:

- normal;
- tree;
- forcedir;
- dfs;
- minbackward;
- maxdepth;
- maxdepthslow;
- maxindegree;
- maxoutdegree;
- maxdegree;
- mindepth;
- mindepthslow;
- minindegree;
- minoutdegree;
- mindegree.

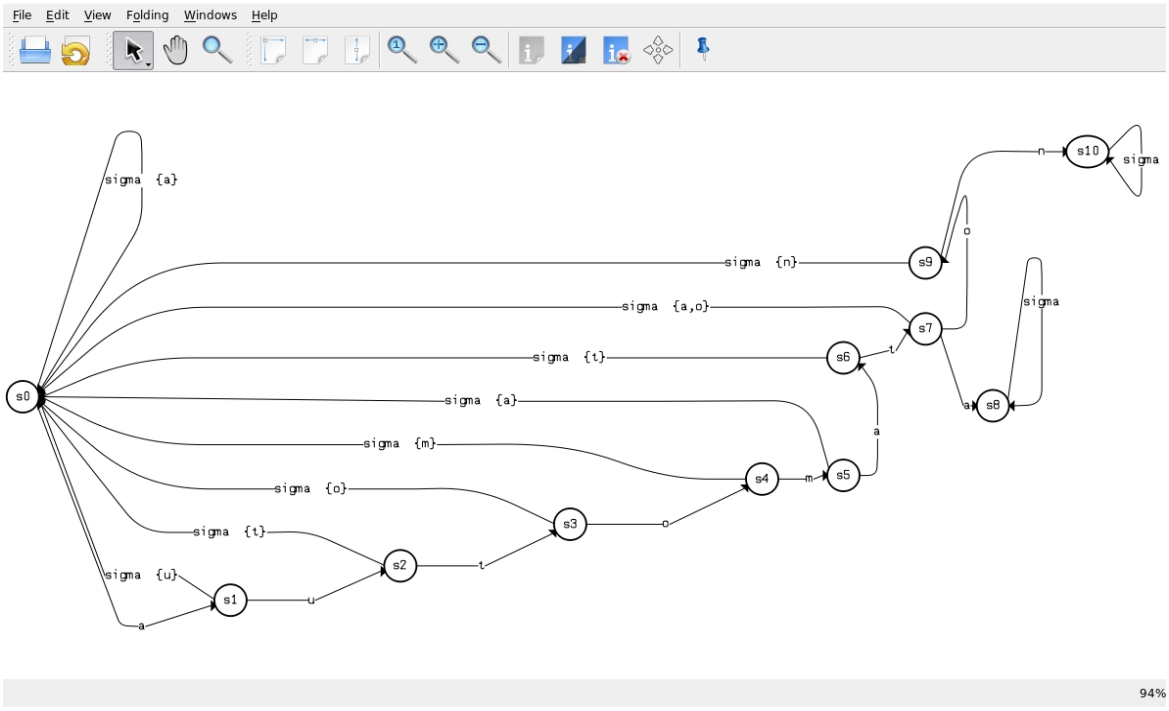


Figure 2.7: Layout of a finite automaton using aiSee.

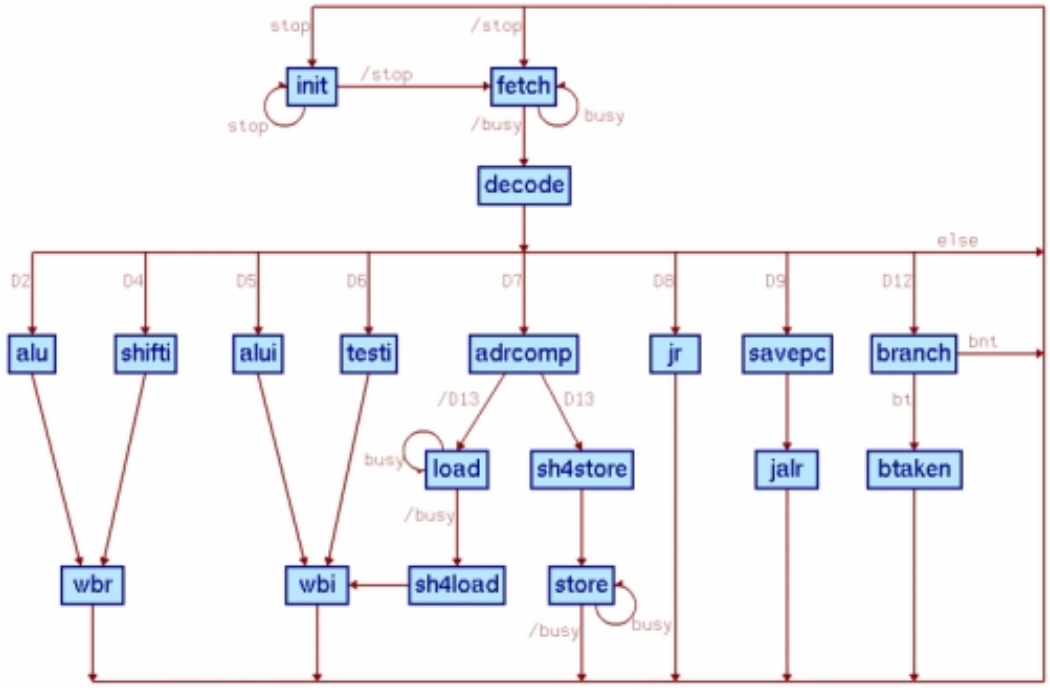


Figure 2.8: Layout of a finite state machine using aiSee.

Initial and final states could not be represented with the usual styles due to style limitations of the nodes. The overall aesthetic of the diagram draw is not bad although the edges, specially self loops, are not handled very well and the labels positioning is not the conventional in finite automata drawings. An example of a finite state machine drawing with manual positioning of states is available at aiSee's website and illustrated in Figure 2.8, but also fails to deliver the aesthetic conventions expected in this type of diagrams.

2.3.2 yWorks

yWorks [yWo10] software provides a set of libraries and applications to visualize class relations in programming languages such as Java, .NET and AJAX. This software is non-free and provides a set of Java class libraries to be used by other softwares. There are three licenses available for this software: single developers, projects and sites. These libraries have layout algorithms for graphs, edges and label positioning. For graph layout the following algorithms are available:

- Tree;
- Circular;
- Hierarchical;
- Organic (force-directed);
- Orthogonal.

It is also possible to just layout the graph edges in which an Organic routing and an Orthogonal routing are available. The label positioning can be done during the graph layout or independently.

There are also available a set of libraries that can export this layouts to various types of formats such as PDF, SWF, EPS, EMF, SVG and GraphML.

yWork was not designed to draw finite automata diagram, although a finite automaton example drawn using yWork layout algorithm is shown in Figure 2.9. First an hierarchical layout algorithm with breadth-first search layering was applied to the graph followed by a force-direct edge routing algorithm and finally a label positioning algorithm. The visible failures in this drawing are the loops that should be rounded, the labels that overlap edges, and the lack of node styles for the initial and final states. The overall result for this example was not bad, taking into consideration that the application was not specifically designed to draw this type of diagrams.

2.3.3 JGraph

The JGraph [jgr10] is a graph drawing component available for Java and prepared to be compatible with Swing. This software is non-free, open-source and distributed under a three clause BSD license. This graph drawing component provides tools to visualize several type of graphs such as process diagrams, workflow visualization, flowcharts, traffic flow, database and WWW visualization, networks, UML diagrams, electronic circuits, VLSI and others. The available layout algorithms are:

- Tree;
- Circle;
- Hierarchical;
- Force-directed.

Figure 2.10 illustrates a finite automaton example drawn using JGraphX (JGraph's graphical interface) with a horizontal hierarchical layout algorithm. The edges' labels positioning is not conventional, self loops are not supported and the initial and final states positioning is not the best.

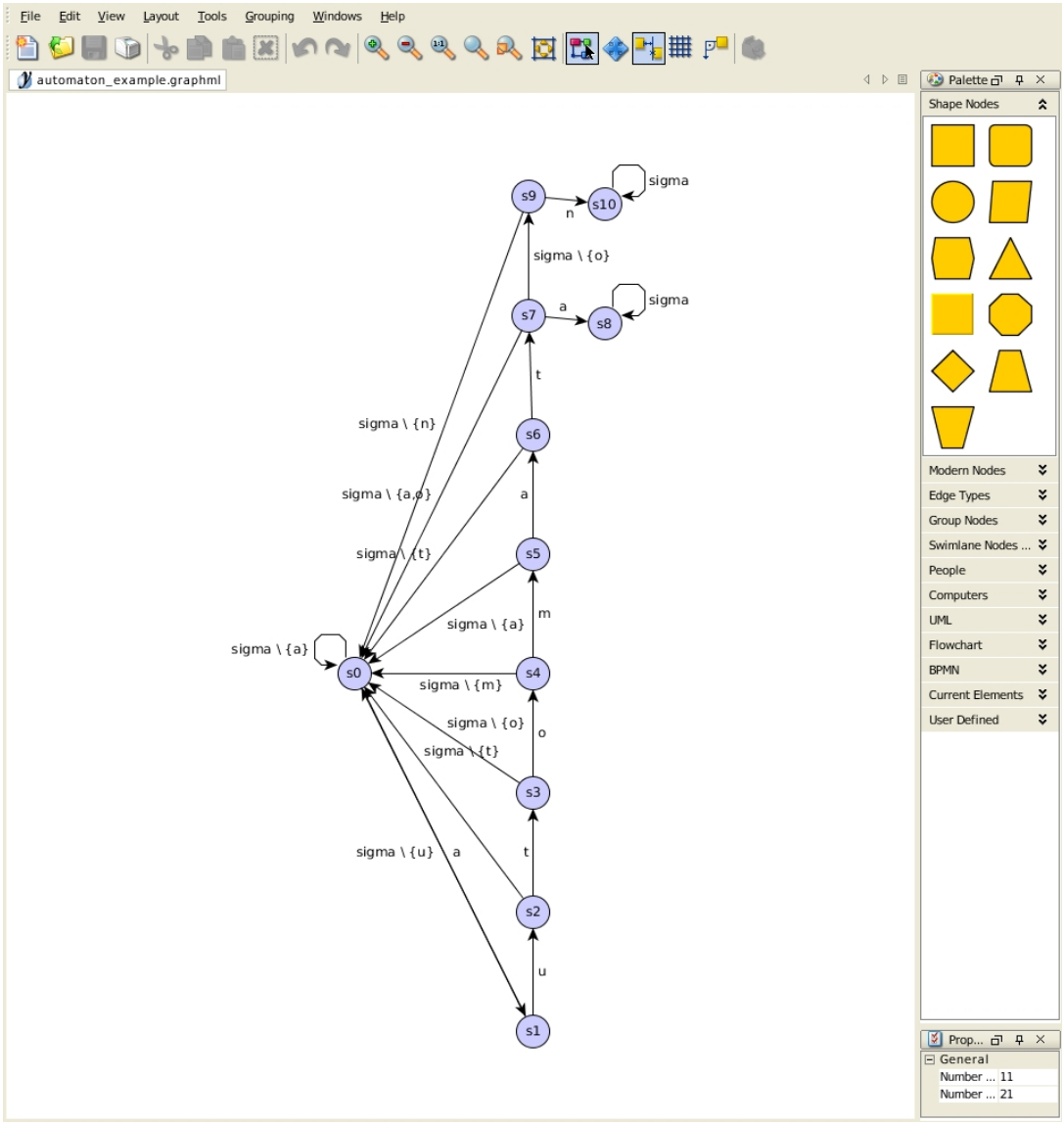


Figure 2.9: Layout of a finite automaton using yWork.

2.3.4 OGDF

The Open Graph Drawing Framework (OGDF) [Tec10] is a project that aims to provide a graph drawing library. This is an open-source library written in C++ and available under the GNU General Public License (GPL). This library implement important graph drawing algorithms such as:

- Biconnectivity decomposition and BC-Tree;

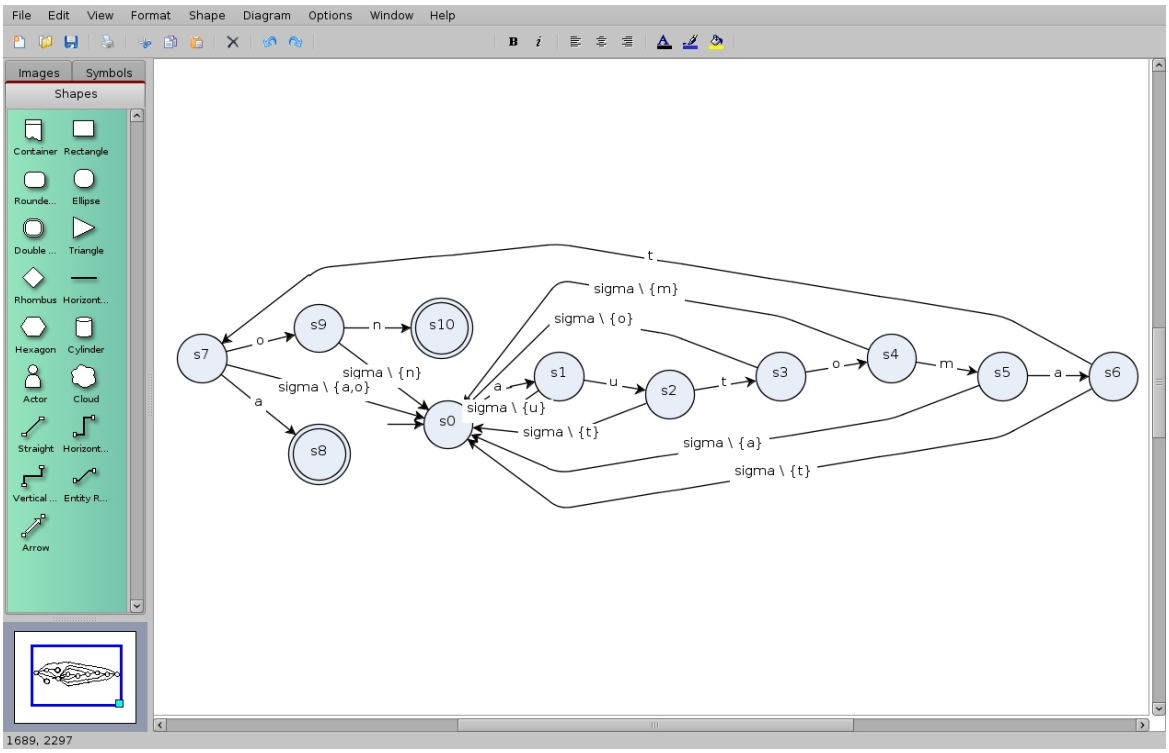


Figure 2.10: Layout of a finite automaton using JGraphX.

- Triconnectivity decomposition and SPQR-Tree;
- Planarity test based on PQ-Trees;
- Planar biconnected augmentation;
- Extration of multiple Kuratowski-subdivisions in linear time;
- NonPlanarCore reduction in linear time;
- Edge insertion with crossing minimization.

These algorithms are the base for the layout algorithms available in this library. The implemented layout algorithms are:

- Orthogonal used for UML and Cluster graphs (UMLOrtholayout and ClusterOrthoLayout);
- Planar (MixedModelLayout, PlanarStraightLayout and PlanarDrawLayout);

- Planarization (UMLPlanarizationLayout, PlanarizationGridLayout and Cluster-PlanarizationLayout);
- Energy-based:
  - FM3 Layout (FMMMLayout);
  - Simulated-Annealing (DavidsonHarelLayout);
  - Fruchterman-Reingold (SpringEmbedderFR).
- Layered (SugiyamaLayout) also for cluster graphs:
  - acyclic subgraph;
  - ranking;
  - 2-layer crossing minimization;
  - coordinate assignment.
- Tree-based (TreeLayout and RadialTreeLayout);
- Misc (CircularLayout and BallonLayout).

The OGDF presents interesting approaches for non-planar graphs that deserve some study.

2.3.5 P.I.G.A.L.E

The Public Implementation of a Graph Algorithm Library and Editor (P.I.G.A.L.E.) [H. 10] is a project that consists in the development of a graph editor and a C++ library that has planar graphs as its main target. This software is open-source and available under GPL. It implements some important algorithms based on new theoretical researches which are:

- Planarity test and embedding (Fraysseix-Rosenstiehl left-right algorithm);

- Kuratowski subdivision or cotree critical partial subgraph extraction (for non-planar graphs);
- Triconnectivity test and decomposition;
- Four-connectivity test;
- Fast Depth-First Search;
- Bipolar and regular orientation algorithms;
- Triangulation of triconnected planar graphs;
- Partitioner based on factorial analysis.

It also provides the following layout algorithms:

- Fraysseix, Pach Pollack algorithm;
- Schnyder algorithm using their triangulation algorithms;
- Schnyder algorithm using a vertex triangulation;
- Tutte barycentric representation of triconnected graphs;
- A Fary representation derived from the Tutte algorithm;
- A spring embedder which preserves the map;
- Visibility representation of planar graphs;
- A drawing of planar graphs using Béziers curves (based on a spring embedder);
- An algorithm to represent biconnected planar bipartite graphs as the incidence graph of horizontal and vertical segments;
- An algorithm to represent planar graphs by contacts of T;
- An algorithm to represent a graph in R^3, as projections of different embeddings of the graph in R^{n-1};

- an heuristic to detect symmetries (experimental);
- an heuristic to find a maximal planar partial graph of a non-planar graph (experimental).

This library introduces new algorithms that can be useful for finite automata drawing but just by themselves do not provide the desired aesthetic for this type of graphs.

2.3.6 Graphviz

Graphviz [LC10] is a well known open-source package in the graph drawing community. This package provides a set of tools for graph drawing purposes, including a C library. There are five command line tools available to draw graphs:

- dot (hierarchical layout);
- neato (spring models);
- twopi (radial layout);
- circo (circular layout);
- fdp (force-directed model).

Graphviz uses its own graph description format called dot [Gra10a]. A set of tools are available to read this format and produce graph drawings that can be exported to several formats including, for example, GIF, JPEG, Portable Document Format (PDF), PS, SVG, GTK Canvas, Xlib Canvas. There are several graphical interfaces for this software as dotty and WebDot. This library is used by several applications to display hierarchical and network type of graphs. Although Graphviz's layout algorithms specially focus in this last two types of graphs, it can be used for several others in particular finite automata diagrams.

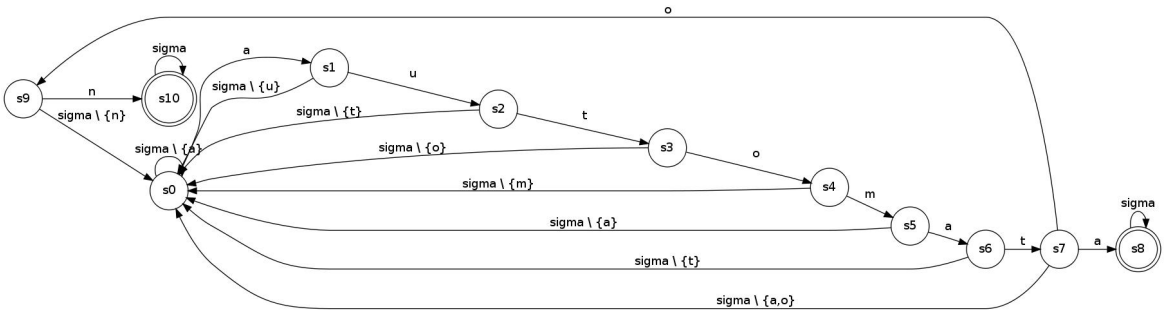


Figure 2.11: Layout of a finite automaton using Graphviz (dot).

Figure 2.11 shows a finite automaton example drawn using Graphviz's hierarchical layout (dot). The result is almost the expected with the exception of four aspects: the positioning of the states *s9* and *s10* that should be more to the right, the edges *(s0,s1)* and *(s1,s0)* that would look better if they were two simple symmetric arcs, the overlap of the *(s0,s0)* self loop's label and the lack of a node style for the initial state. Besides these minor problems, the resulting layout for this specific diagram is good, although more complex diagrams may be a problem for a hierarchical layout. The spring model also presented an interesting layout, but could not provide a left to right readability.

2.3.7 JFLAP

JFLAP [RF06] is an application that aims to provide a way to experiment with formal languages representations such as nondeterministic finite automata, nondeterministic pushdown automata, multi-tape Turing machines, several types of grammars, parsing, and L-systems. This software is open-source and distributed under the JFLAP 7.0 license. It provides a finite automata editor that allows the user to draw its finite automata and manipulate them. It has implemented some basic layout algorithms:

- Tree (degree and hierarchical);
- Generalized Expectation-Maximization (GEM);

- Circle;
- Two circle;
- Spiral;
- Random.

Clearly JFLAP do not focus its work on the layout of the finite automata, thus, the available layout algorithms are very basic and simple. Figure 2.12 illustrates an automaton drawn using JFLAP's tree degree layout algorithm which has been the one that presented the most pleasant layout for this example. This layout has no edge crossing although the edge's labels overlap and make it very difficult to read. The left to the right readability also fails, but in this particular case a simple ninety degrees rotation would solve that problem.

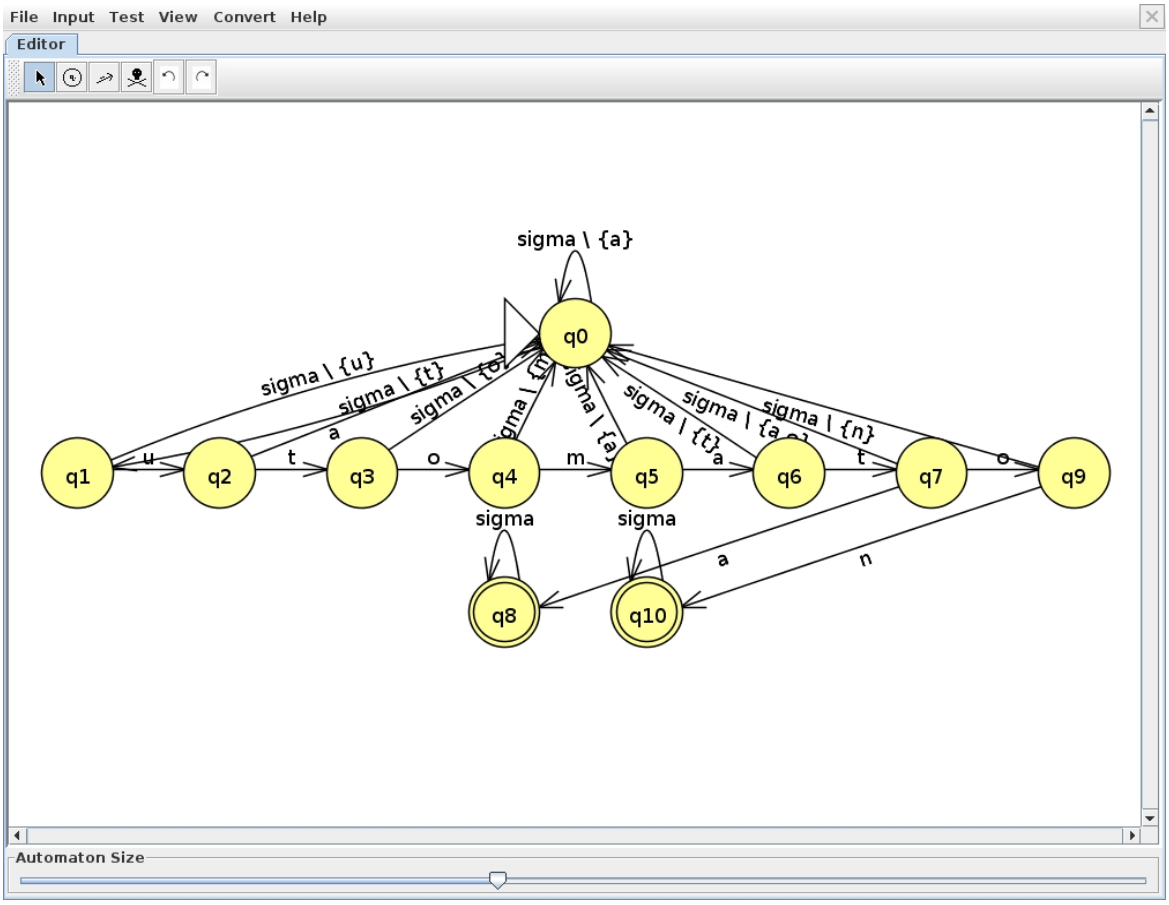


Figure 2.12: Automaton example using JFLAP's two circles layout.

Chapter 3

GUItar

GUItar [FAd10a, AAA+09] is a project that aims to provide a graphical environment for finite automata visualization and editing. Although GUItar specially focuses in finite automata type of diagrams, it supports many types of diagrams due to its versatility, such as transducers and Turing machines. An example of a Turing machine drawing on GUItar is shown in Figure 3.1. Currently GUItar is implemented in Python [Fou10] and uses wxPython graphical toolkit. Its canvas is implemented using the wxPython's FloatCanvas [Bar10] module. The graphical interface basic frame is composed by a menu bar, a tool bar and a notebook. Both the menu bar and the tool bar are dynamically built from XML configuration files, which provides an easy way to configure them. The notebook handle multiple pages, each one containing a canvas.

3.1 GUItar's Canvas

GUItar is an application mostly mouse driven, and most of its interfacing is done through the canvas. As said before, the canvas uses the wxPython's FloatCanvas module which provides a set of graphical objects that can be bound with mouse events. The bound mouse events are defined by the selected mouse mode. All mouse modes are created by extending a base mode and redefining some of its mouse event bindings.

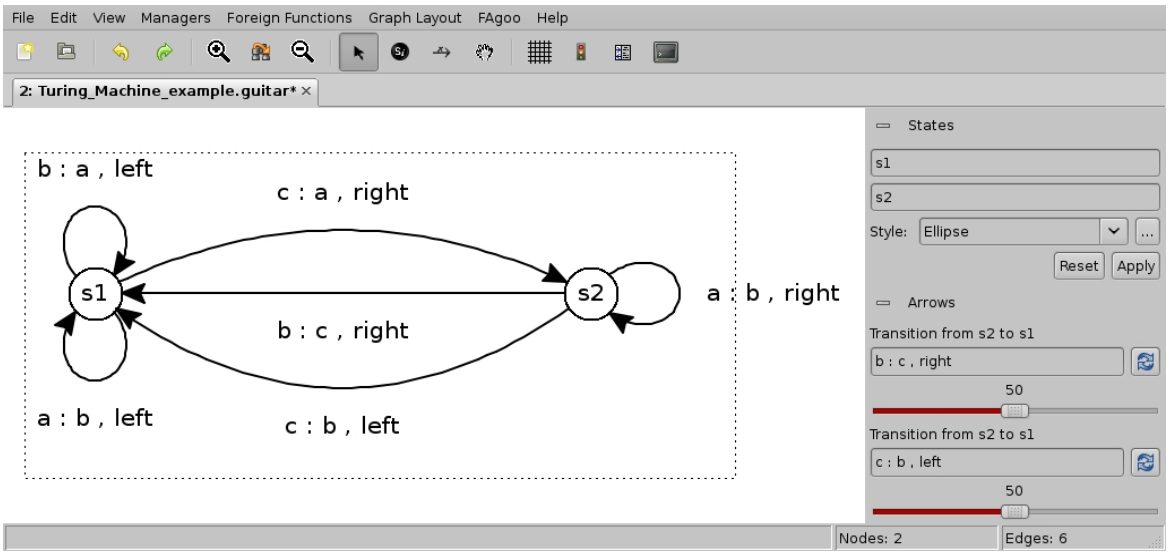


Figure 3.1: An automaton created in GUItar.

There are four mouse modes available:

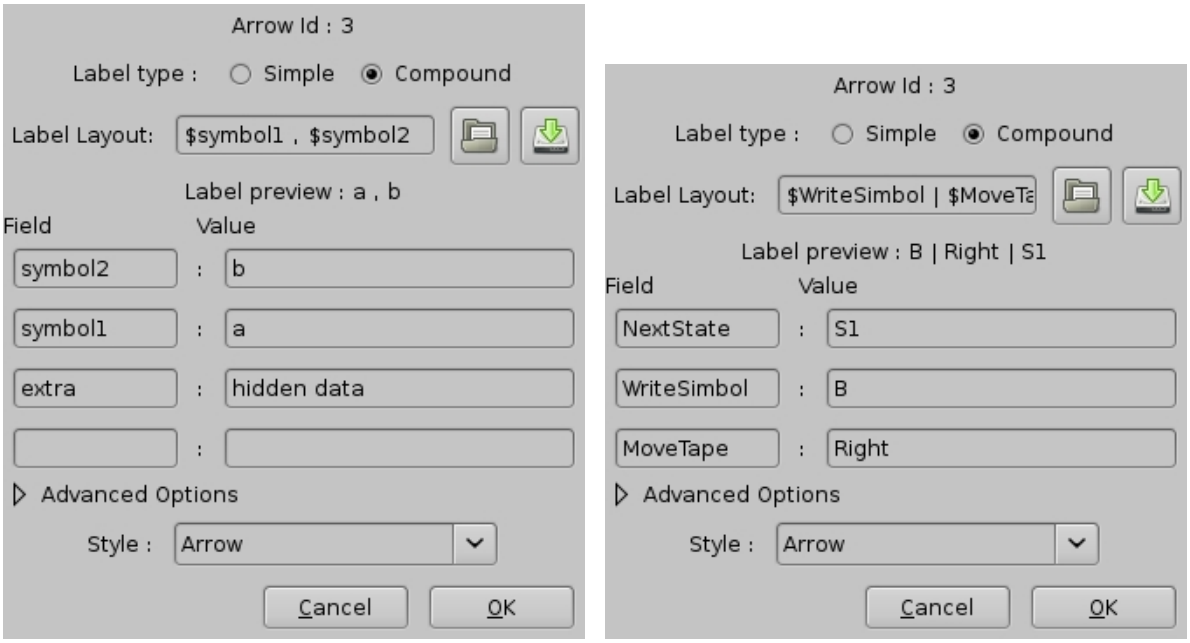
- Mouse mode (to edit the existing objects);
- Node mode (to add nodes);
- Arc mode (to add arcs);
- Move mode (to freely move the diagram).

The existing *Spline* object in FloatCanvas did not provide enough information about the spline to correctly place the arc labels. Therefore, a new object, *ArrowSpline*, was implemented in order to correctly draw the arc labels. To draw the diagrams' nodes, arcs and labels the following, FloatCanvas primitive objects are used:

- *Rectangle* (nodes);
- *Ellipse* (nodes);
- *ArrowSpline* (nodes and arcs);
- *ScaledText* (labels).

The nodes are adjusted into a grid to avoid overlaps. If a node is dropped above another, then the closest free position is found to accommodate it. The arcs, when added, are automatically arranged in order to keep the arcs from the left to the right above the ones from the right to the left. It is also possible to move the arc's control points in stepwise movements. The self loops are a particular type of arcs, and therefore, treated differently. The self loop control points are found by intersecting a circle with the node. There is also a special control point in the center of the loop, that basically controls the circle that is used to calculate the control points.

The node and arc labels can be either simple or compound. Simple labels are just text strings, while compound labels have custom fields with values specified by the user. The user can choose either to display each label field or not and, this way, extra data can be associated to nodes and arcs (see Figure 3.2).



(a) (b)

Figure 3.2: Two examples of compound labels.

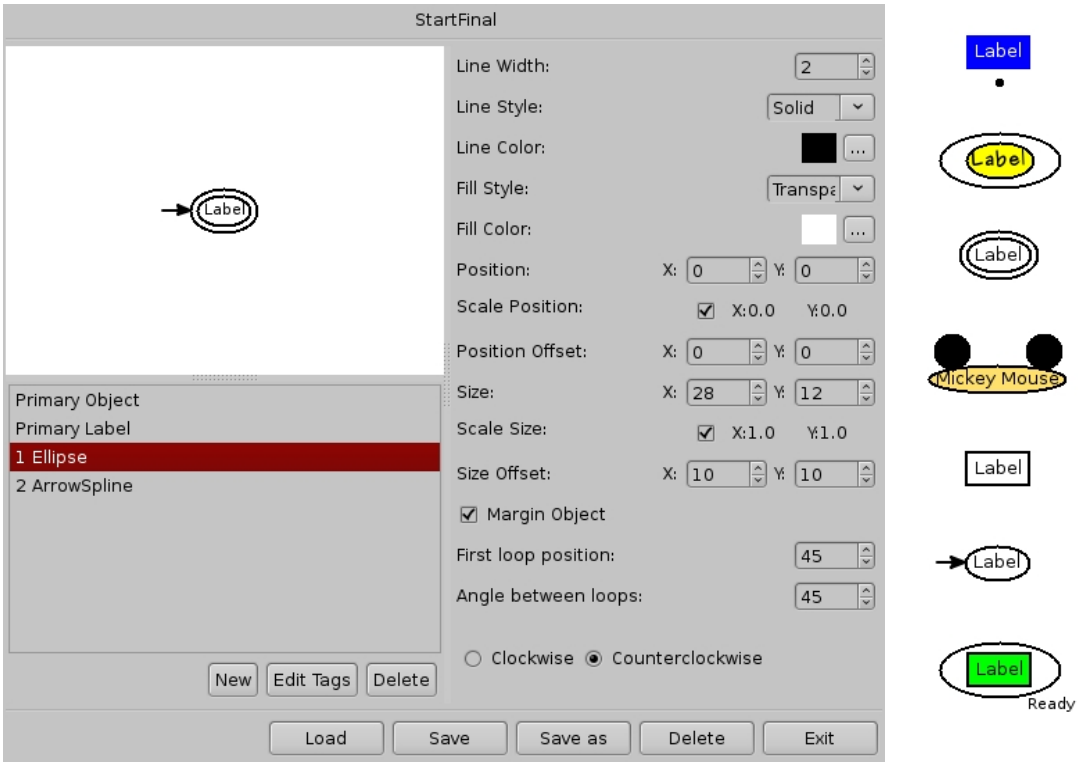
3.2 Styles

GUItar provides a node and arc style manager that not only allows the management of multiple styles, but also provides an interactive way to edit and create new ones. The graphical representation of a node consists in a set of objects. The available objects for nodes are ellipses, rectangles, arrowed splines and scaled texts. Each node must have at least an ellipse or a rectangle to ensure that it has a place to dock the incoming and outgoing arcs. It must also have one scaled text to place the node's label. This node structure allows the creation of complex nodes, enriching the graph visualization. For example in finite automata diagrams we can represent final states by using two concentric ellipses, or initial states using an arrow pointing to an ellipse. Figure 3.3a shows the GUItar's node style manager, editing a style. A few node styles created using this style manager are shown in Figure 3.3b.

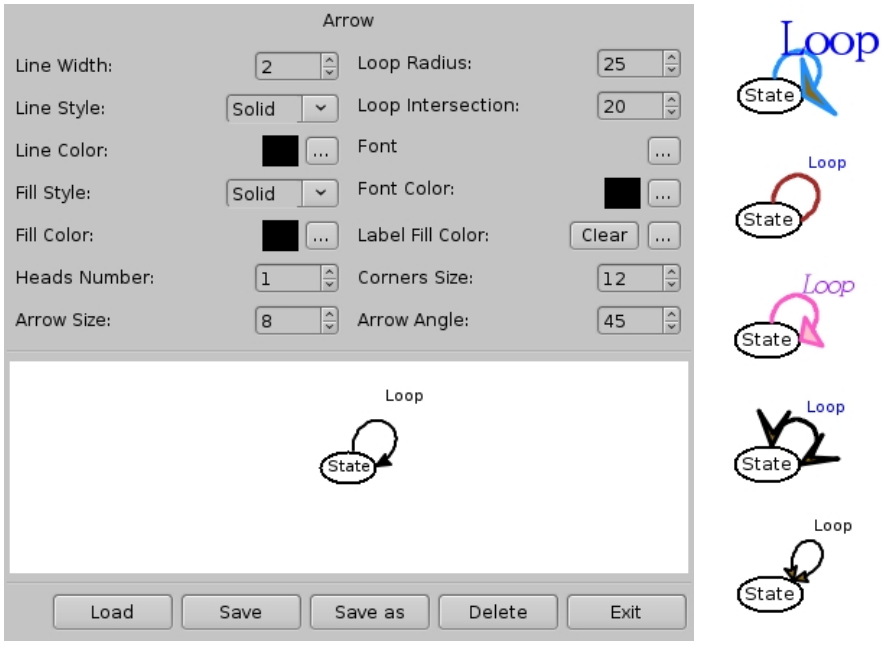
The arc style manager allows the editing and creation of rich arrow styles. A large set of properties, such as arrow heads number, arrow head size, arrow head angle, line width, line color, fill color and label font, are available. It is also possible to define default behaviours for loops according to its style. The arc style manager and some examples of arcs styles are presented in Figures 3.3c and Figure 3.3d, respectively.

3.3 FFCs

The GUItar project was not intended to be a new monolithic graph visualization and editing tool, but supposed to be seen more as a hub where graph manipulation libraries can, together, provide better visualization and manipulation tools. This is achieved by a FFC mechanism, using a Python interface to access the external tools (see Figure 3.4). There are three types of FFC: module FFC, object FFC and interactive FFC. In the first case, the FFC calls a function directly from an external Python module. In the second case, it creates a foreign object and then calls methods of that object. In the last case, when a specific event occurs, the FFC triggers the respective handler function from



(a) Node style manager. (b) Node styles.



(c) Arc style manager. (d) Arc styles.

Figure 3.3: GUItar's style managers and a few styles examples.

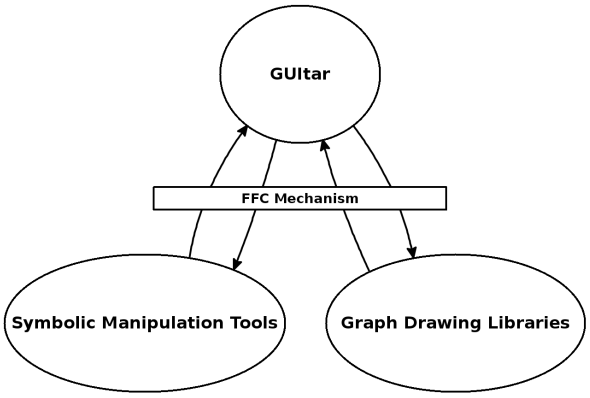


Figure 3.4: A FFC mechanism overview.

an external Python module that will return a sequence of actions as script commands. FFCs require an XML configuration file that specifies the available methods. FFCs can create their own menu entries which makes its integration in GUItar smooth and practical. Most of the GUItar tools are implemented using FFCs such as the interfaces of FAdo and FAgoo.

3.4 Graph Classification

The GUItar classification mechanism allows to test if a graph belongs to a certain class by checking if the graph verifies a set of properties. These properties can test graphical properties (e.g. if arcs have arrows) or semantic properties (e.g. if a finite automaton is deterministic). A few of these methods are predefined in GUItar to check the most usual graphical properties of a graph. Access to external libraries with FFCs can be used to test graph properties, broadening the class range. Biconnectivity and planarity tests can be done, for example, using FAgoo.

A friendly interface is available for graph classification (see Figure 3.5). This interface lists the graph properties and identifies the ones that are verified in the current graph. The user can create his own classes by stating the properties that the class must comply. It is also possible to export and import these class definitions.

Figure 3.5: GUItar's interface for graph classification.

3.5 Semaphores

When editing a graph it can be useful to constraint the actions performed so that the resulting graph does not leave a certain class. The GUItar *Semaphore* tool assists this task by warning the user, or even restricting his actions. For example, suppose that we have a deterministic finite automaton (DFA) as the result of some manipulation, and we want to edit it. We can enable the *semaphore* for DFAs to ensure that the changes that we apply to the graph do not compromise the DFA class definition.

New *Semaphores* can be created by extending the *Semaphore* base class and declaring them in an XML configuration file. An image of a traffic sign is associated to each *semaphore* whose light color represent the current state of the graph evaluation. There is also an image of a small padlock that when closed means that actions are restrict,

i.e., do not allow actions that compromise the desired graph properties.

3.6 Import and Export

GUItar store its graphs using GUItarXML [AMR10], which is an XML format specially designed for this application and based on GraphML [Gra10b]. GUItar also imports and exports to other formats, converting from and to GUItarXML. Currently the available exporting formats are GraphML, dot [Gra10a], Vaucanson-G [LS09] and FAdo. It is also possible to import from all these formats with the exception of Vaucanson-G. The Xport mechanism provides an easy way to add new export and import methods to GUItar.

Chapter 4

FAgoo Implementation

Since GUItar is implemented using Python and its extension mechanism interface, the FFC, is Python oriented, FAgoo either had to be implemented using Python or provide a Python interface to it. Python is a high level language that allow the easy and fast development of large applications in a small amount of time, although for performance reasons it was decided not to use Python for this library implementation. Python is implemented in C and its API provides tools to create new built-in modules written in C or C++. Thus, FAgoo is implemented as a C extension of Python, providing this way a high-level interface for this library, that can be easily interfaced with GUItar using FFCs, and having at the same time the performance of a C library. FAgoo is being developed so that with minimal modifications it could be separated from Python's API and used directly as a C library.

4.1 Conventions and Aesthetics

Graph drawing algorithms usually are specialized for certain classes of graphs, i.e., they check for specific properties before its application, for example an algorithm to draw a graph without edge crossing requires a planar graph as input. Generally graph drawing algorithms require the input graph to belong to one or more specific classes.

This is done because the algorithms can only work with that type of graphs (presents better results) or that type of graphs are expected to be handle differently.

The *drawing conventions* of a graph are the basic properties that are expected to be satisfied in a drawing. The drawing conventions of finite automata were described in Chapter 2.2 and some of them are complex and not easy to achieve. The following drawing conventions are widely used in graph drawing:

- Grid Drawing (vertices, edges bends and crossings have integer coordinates, see Figure 4.1a);
- Polyline Drawing (edges are drawn as polygonal chains, see Figure 4.1b);
- Orthogonal Drawing (edges are drawn as polygonal chains of vertical and horizontal segments, see Figure 4.1c);
- Upward Drawing (for digraphs where edges are drawn as curves monotonically non-decreasing in the vertical direction, see Figure 4.1d);
- Straight-Line Drawing (edges are drawn as a straight line segments, see Figure 4.1e);
- Planar Drawing (edges do not cross, see Figure 4.1f).

The straight-line drawing is a particular case of polyline drawings and this in turn can be planar making it a planar straight-line drawing. These drawing conventions are usually used as base for more complex drawing conventions.

There are graphical properties that improve the graph readability. These properties are called *aesthetics* and some frequently adopted are:

- *Crossings* (Minimize the number of edges crossings. If possible have planar drawings.);
- *Overlaps* (Avoid the overlap of edges with edges' labels. Finite automata can have large edge labels.);

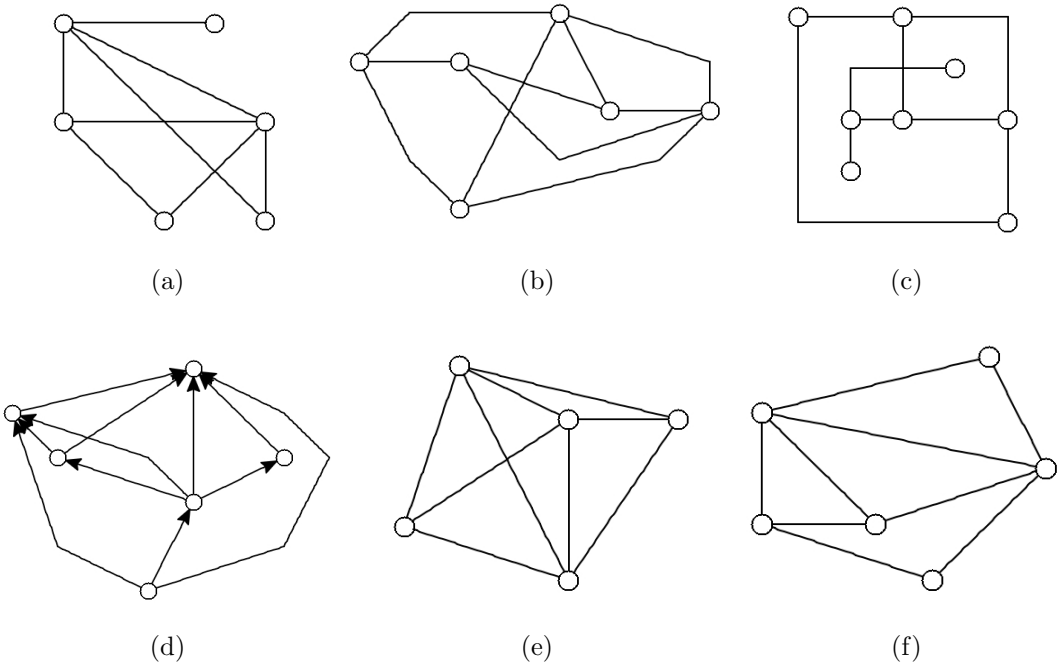


Figure 4.1: Some examples of drawing conventions.

- *TotalEdgeLength* (Minimize the sum of the edges' length.);
- *MaximumEdgeLength* (Minimize the maximum length of each edge.);
- *UniformEdgeLength* (Minimize the length difference of the edges.);
- *TotalBends* (Minimize the total number of bends along the edges. If possible have a straight-line drawing.);
- *MaximumBends* (Minimize the maximum number of bends on an edge.);
- *UniformBends* (Minimize the variance of the number of bends on the edges.);
- *AngularResolution* (Maximize the smallest angle between two edges incident on the same vertex.);
- *Area* (Minimizing the area of drawing.);
- *AspectRatio* (Minimize the aspect ratio of a drawing.);
- *Symmetry* (Reveal the symmetries of the graph in the drawing.).

The area aesthetic is relevant because usually the visualization area is limited and some times small. For example, the visualization of networks can be difficult even in the largest screens due to the generally large size of these graphs. Another example is the representation of diagrams in paper (e.g. finite automata diagrams, or dataflow diagrams). The area of drawing is usually formally defined as the area of the smallest rectangle that covers the drawing this is so because the visualization support is usually rectangular (e.g. monitor, and paper sheets). But the area can be defined differently, for example as the area of the smallest convex polygon that cover the graph. The *aspect ratio* of a graph is defined by the ratio between the length of the longest and the shortest side of the rectangular area of the drawing. The aspect ratio is related with the area of the drawing because the balance between these two aesthetics is very important. For example, a drawing with a small drawing area and a high aspect ratio, can not be well visualized on a screen. The ideally would be having a small area and small aspect ratio.

Most of these aesthetics give rise of computationally hard problems, so many different strategies and heuristics approximations have been developed.

4.2 Implementation Choices

As said before, graph drawing algorithms usually take as input a graph satisfying a certain set of properties. It is often thanks to this restriction that it is possible to find solutions that approximate the polynomial execution time. It is frequent to have an input graph that does not meet the required properties of the drawing algorithm. So, when possible, the graph is first transformed in order to meet these properties, then the drawing algorithm is applied and finally it is transformed back to its original form. For example, graph drawing algorithms usually assume that the input graph is connected, so if the input graph is disconnected, edges are added in order to make the graph connected, and after applying the drawing algorithm, the previously added edges are removed. Bertolazzy et al [BBL95] presented a taxonomy where graph classes

and graph drawing algorithms are represented hierarchically. Each graph drawing algorithm is defined on a graph class, and inherited by all the subclasses. A taxonomy of the graph classes addressed by FAgoo is illustrated in Figure 4.2a, and Figure 4.2b uses the Bertolazzy et al taxonomy to represent the current implementation path of FAgoo and a possible one for the future.

At this early stage of FAgoo's implementation, it was necessary to implement many basic graph manipulation algorithms such as connectivity and biconnectivity augmentation algorithms. The currently implemented layout algorithm is a straight-line drawing, that draws planar graphs without edge crossings and uses only straight-lines to draw the edges. This algorithm requires that the input graph is planar and triangulated. Kant [Kan93] presents an algorithm to triangulate a planar graph

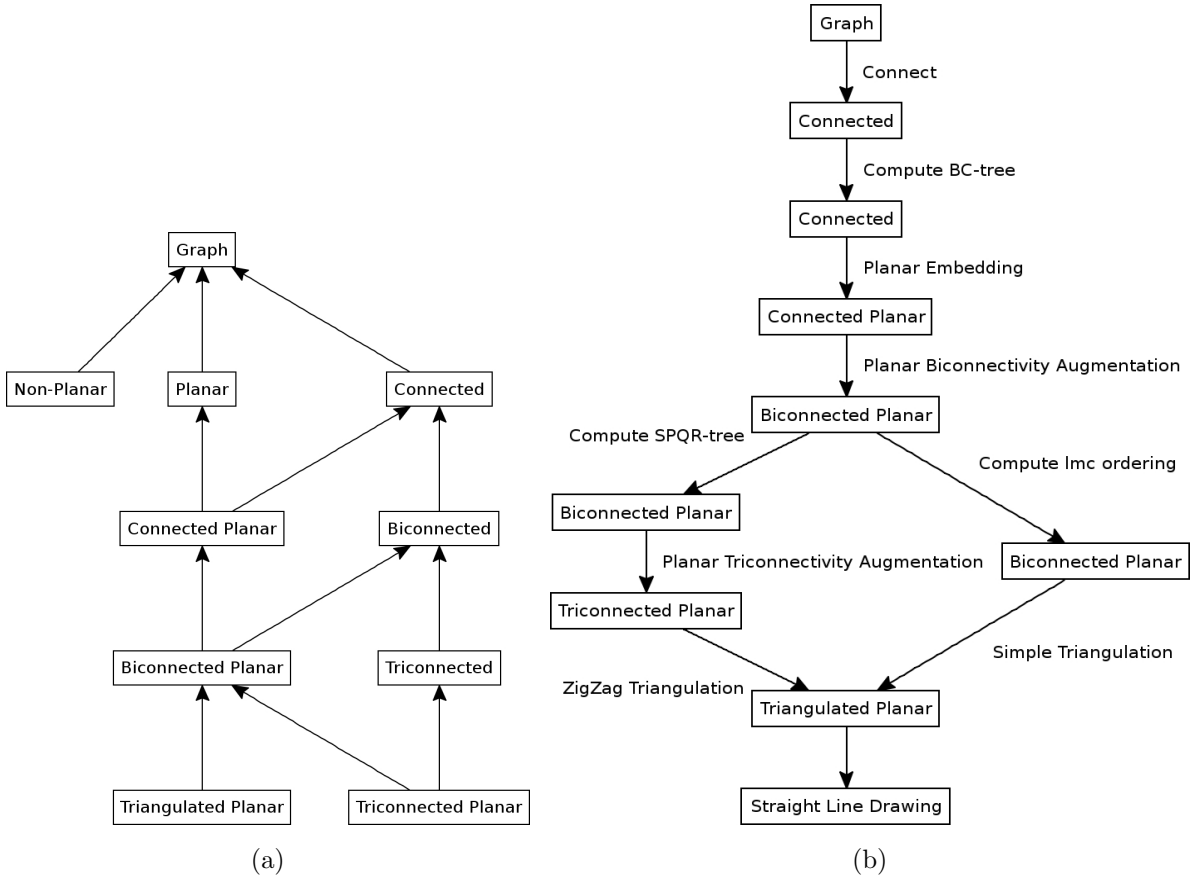


Figure 4.2: A taxonomy of FAgoo's graphs and its implementation path.

while minimizing the maximum degree which is a good approximation of the optimal solution. Although these algorithms requires the input graph to be triconnected and FAgoo currently does not yet implement triconnectivity augmentation algorithms. So another triangulation algorithm was implemented, which only requires the input graph to be biconnected and as side effect computes a leftmost canonical ordering (lmc-ordering) [Kan93, 127] that is needed for the straight-line drawing algorithm.

4.3 Implementation Programming Language

FAgoo intends to be a graph drawing library for Python, because Python is a high level language, object oriented and very easy to use. This way FAgoo can easily be embedded in other applications. To implement FAgoo as a Python module there were considered three possibilities:

- write it in Python;
- write it in C and then use an automatic wrapper and interface generator (e.g. SWIG [Dav10] and Boost.Python [Boo10]) to export it to Python;
- write a extension module for Python in C.

Writing FAgoo in Python was excluded due to performance reasons, and since Python provides such good support for writing C modules, FAgoo is currently implemented as a C module for Python. This combination allows the creation of an efficient graph drawing library, available to a high level language, making its usage very easy and practical.

4.4 SimpleGraph Object

FAgoo provides a Python object named SimpleGraph, which implements a set of methods for graph manipulation. This object handles the input graphs as multigraphs,

i.e., undirected graphs that allow multiple edges. Its constructor takes a ElementTree, a string of GUItarXML (see Chapter 3.6) or a Python list with pairs of integers representing edges. The ElementTree is an object from the lxml Python module, that provides methods to load XML files as trees of Element objects, and save them back. The XML specification accepted is GUItarXML that is used by GUItar. The graph is maintained in a adjacency list and each vertex and edge contains a pointer to its respective Element object, to be used when extra information is needed. The SimpleGraph object implements the following set of methods for its manipulation:

- GetEdgesNumber (Returns the number of edges in the graph);
- GetNodesNumber (Returns the number of vertices in the graph);
- GetGraphEtree (Returns the graph as a lxml Element object);
- GetGraphAsGuitarEtree (Returns the graph as a lxml ElementTree object);
- AddNode (Add a new vertex to the graph);
- AppendNode (Add a new vertex to the graph from an Element object);
- GetNodeEtree (Returns a vertex Element object representation);
- AddEdge (Add a new edge to the graph);
- AppendEdge (Add a new edge to the graph from an Element object);
- GetEdgeEtree (Returns the edge Element object representation);
- IsPlanar (Returns true if planar, otherwise false);
- IsConnected (Returns true if the graph is connected, otherwise returns false);
- MakeConnected (Makes the graph connected by adding edges between the connected components);
- IsBiconnected (Returns true if the graph is biconnected, otherwise returns false);

- GetBicomponents (Returns the graph biconnected components as a list of SimpleGraph objects);
- GetBCTree (Returns a SimpleGraph object with a graph representation of its BC-Tree);
- MakeBiconnected (Returns true if the graph is planar and was successfully made biconnected, otherwise returns false);
- IsTriangular (Returns true if the graph is triangular otherwise returns false);
- Triangulate (If the graph is planar, triangulates it, otherwise leave it unchanged);
- SetSLDraw (If the graph is planar, set its layout to a straight-line draw, otherwise leave it unchanged).

4.5 Biconnectivity

FAgoo implements algorithms to test if a graph is biconnected, to compute its *BC-tree*, and to augment it so that it becomes biconnected. The algorithm to test biconnectivity and compute the *BC-tree* is an adaptation of the one presented by Hopcroft and Tarjan [HT73]. The original algorithm tests if the graph is biconnected in linear time. To explore the graph, this algorithm uses a depth-first search. For a graph $G = (V, E)$, it starts at a vertex v of G and an unexplored adjacent vertex u is chosen to proceed the search. The vertex v is called the *parent* of u. When the depth-first search finishes on the vertex u, it proceeds on the next unexplored adjacent vertex of v. If the depth-first search is applied to each connected component of G, then each node will be visited exactly once. The algorithm uses a depth-first search to number the vertices from 0 to n, where $|V| = n$. A vertex u is a cutvertex, if all the vertices that were explored as result of the depth-first search on u, have a higher numbering than it.

The algorithm uses a structure (*BC_Vars*) to keep a stack with the traversed edges and the graph BC-tree (*BC_Tree*). During the depth-first search the traversed edges are

added to the stack and when a cutvertex u is found, all the edges on the stack added after the edge (v, u), where v is the parent of u, form a biconnected component. Each biconnected component originates a B-vertex, which is connected to the respective C-vertices and added to the BC-tree.

The function $BDFS$ receives four arguments: a BC_Vars structure $vars$, thet keeps all the necessary auxiliary variables such as the edge stack and the BC-tree, the current vertex v, the id of the parent vertex, and the variable *lowpoint* with the number of the lowest vertex reached. The first value of *lowpoint* is $|V|+1$ and it is updated through the function execution.

```
int
BDFS(BC_Vars *vars, BC_Vertex *v, int parentid,int lowpoint){
  BC_CVertex_List *neighbors=NULL; BC_Edge_List *bc_block=NULL;
  int newlowpt; BC_Vertex *w;
```

For each edge (v, w), the function tests if (v, w) was not traversed, by testing if the number of w is lower then the number of v, and if w is not parent vertex of v. If it was not traversed then it is added to the edge stack.

```
  BC_Edge *edge=v->firstedge;
  while(edge!=NULL){
    w=edge->target;
    if(w->number < v->number && w->id!=parentid){
      vars->edgestack[vars->ep]=edge; vars->ep++;
```

At this point the vertex w was either not visited yet or the edge (v, w) is a back edge, i.e., the number of w is lower then the number of v. If the vertex w was not visited yet then it is numbered and the function is recursively called for the vertex w. Then the *lowpoint* variable is updated with the minimum between it self and the returned lowest point.

```
      if(w->number==0){
  vars->vertexnumber++; w->number=vars->vertexnumber;
  newlowpt=BDFS(vars,w,v->id,vars->V+1);
  lowpoint=min(lowpoint,newlowpt);
```

If v is a cutvertex then the returned lowest point must be greater or equal then the number of v, in which case all in the edge stack above the edge (v, w) form a biconnected component. If v has no parent, i.e., it is the vertex of the first call, then it is a cutvertex only if there is another biconnected component to be explored.

```
if(newlowpt >= v->number){
  int becvertex=1;
  if(parentid<0 && v->cvertex==NULL){
    BC_Edge *rest=edge->next;
    becvertex=0;
    while(rest!=NULL){
      if(rest->target->number < v->number){
becvertex=1;
break;
      }
      rest=rest->next;
    }
  }
  int cutvertex=0;
  neighbors=NULL;
  if(becvertex){
    if(v->cvertex==NULL){
      BC_CVertex_New(v,NULL);
      BC_Tree_AddCVertex(vars->bctree,v->cvertex)
    }
    neighbors=BC_CVertex_List_Push(NULL,v->cvertex);
    cutvertex++;
  }
```

Each vertex maintains a pointer to the respective C-vertex of the BC-tree. To create the B-vertex that represents the biconnected component, the edges on the stack above the edge (v, w) and it-self are pop from the stack a added to a list on the B-vertex. The list *neighbors* keeps all the C-vertices on the biconnected component.

```
  bc_block=NULL;
  int neigbors_count=0;
```

```
    BC_Edge *aux=vars->edgestack[vars->ep-1];
    while(aux->source->number > v->number){
      if(aux->source->cvertex!=NULL){
        neighbors=BC_CVertex_List_Push(neighbors,aux->source->cvertex);
        cutvertex++;
      }
      if(aux->target->cvertex!=NULL){
        neighbors=BC_CVertex_List_Push(neighbors,aux->target->cvertex);
        cutvertex++;
      }
      bc_block=BC_Edge_List_Push(bc_block,aux);
      neigbors_count++;
      vars->ep--;
      aux=vars->edgestack[vars->ep-1];
    }
    if(w->cvertex!=NULL){
      neighbors=BC_CVertex_List_Push(neighbors,w->cvertex);
      cutvertex++;
    }
    bc_block=BC_Edge_List_Push(bc_block,edge);
    neigbors_count++;
    vars->ep--;
    BC_BVertex *bvertex=BC_BVertex_New(0, bc_block,neigbors_count,
                neighbors,cutvertex);
```

Finally the C-vertices on *neighbors* list are added as neighbors of the created B-vertex, and the B-vertex is added to the BC-tree.

```
    while(neighbors!=NULL){
      BC_CVertex_AddNeighbor(neighbors->cvertex,bvertex)
      neighbors=neighbors->next;
    }
    BC_Tree_AddBVertex(vars->bctree,bvertex)
  }
      }
```

In the case of (v, w) being a back edge, the *lowpoint* variable is updated with the minimum between it-self and the number of w.

```
      else lowpoint=min(lowpoint,w->number);
    }
    edge=edge->next;
  }
 return lowpoint; }
```

To test if a graph is biconnected, this algorithm is applied to each of its connected components, and if the BC-tree has only one B-vertex, then the graph is biconnected. To connect a graph, its BC-trees are computed and each pair of BC-trees are connected by adding an edge between the C-vertices with lower degree of each component. The C-vertices are used to connect the BC-trees to avoid create more C-vertices.

Figure 4.3 and Figure 4.4 show a graph example and its BC-tree, respectively.

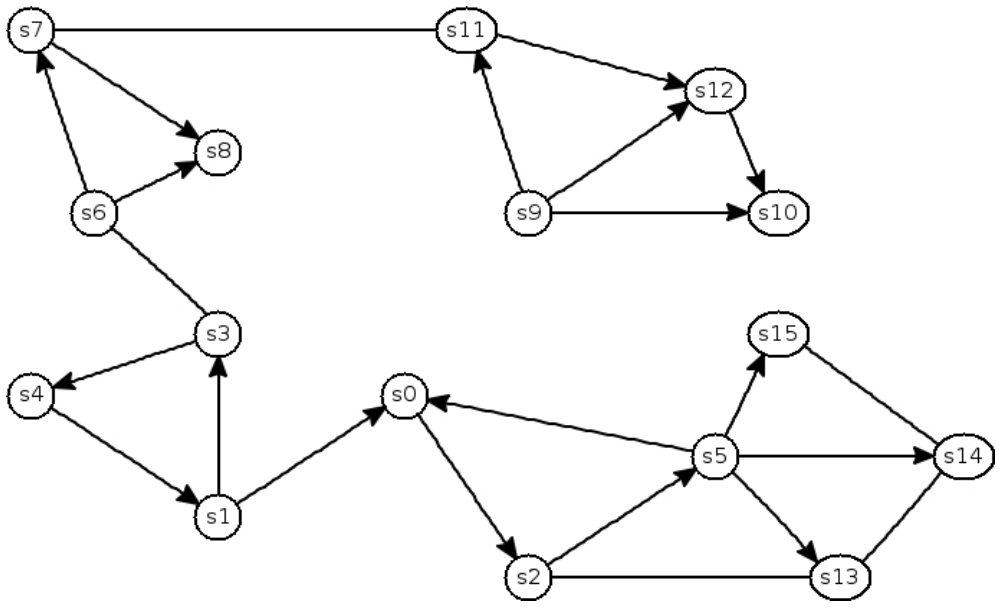


Figure 4.3: Example graph G_1.

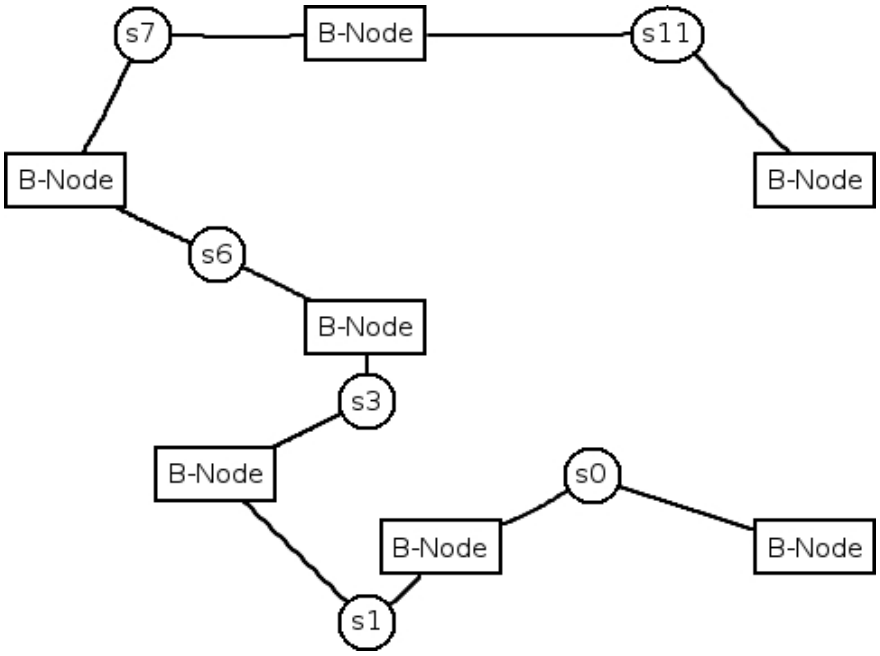


Figure 4.4: The BC-tree of G_1.

4.6 Planarity testing and embedding

Planar graphs have a major role in Graph Drawing, because of its important drawing properties. But even non-planar graphs are usually transformed into planar graphs (by removing edges) in order to be better drawn. The Kuratowski's theorem [Kur30] states that a graph G is planar if and only if it has no subgraph homeomorphic to $K_{3,3}$ or K_5 ($K_{3,3}$ is the complete bipartite graph on 2 sets of 3 vertices and K_5 is the complete graph with 5 vertices, see Figure 4.5). Although, this approach is not practical and therefore other techniques for planarity testing where developed. There are, several algorithms for planarity testing which are based on one of two principles, the *edge addition* and the *vertex addition*. The edge addition algorithm was originally presented by Auslander and Parter [AP61] and correctly formulated by Goldstein [Gol63]. A linear time implementation of this algorithm was developed by Hopcroft and Tarjan [HT74]. FAgoo currently implements this algorithm for planarity testing and to compute a planar embedding it implements the algorithms presented by Mehlhorn and Mutzel [MM96].

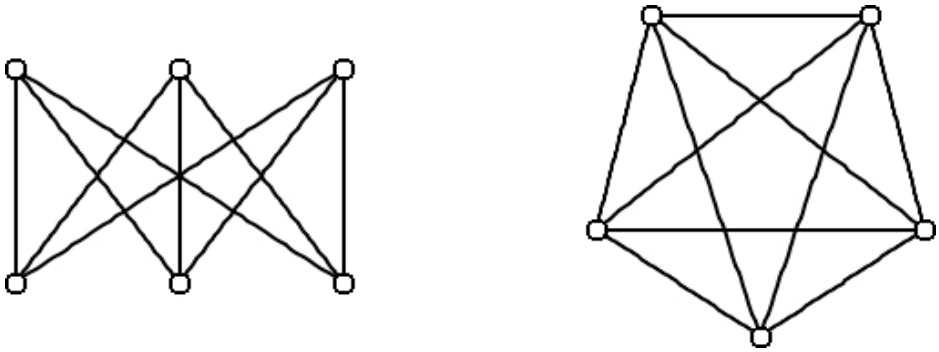
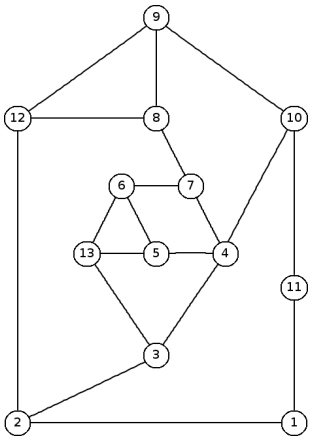


Figure 4.5: On the left the $K_{3,3}$ graph and on the right the K_5 graph.

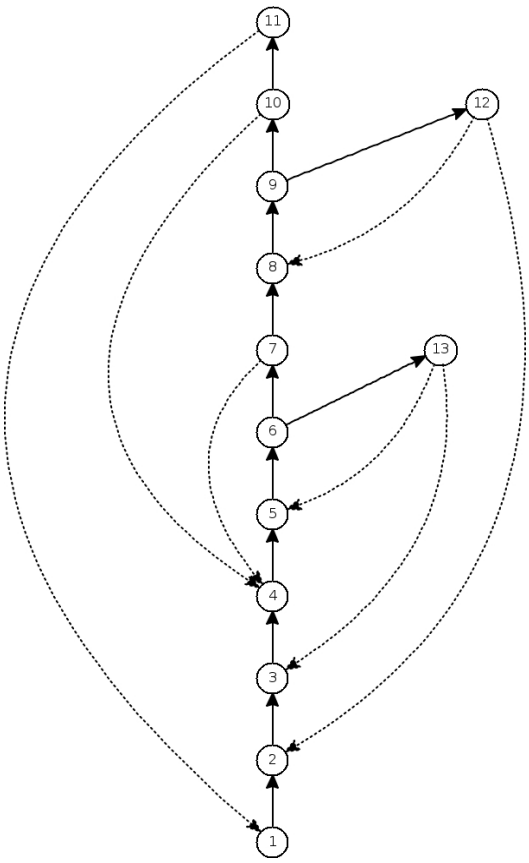
To test if a graph $G = (V, E)$ is planar, the algorithm starts by testing if the number of edges of G exceeds $3n-3$, where $|V| = n$. If that test fails then the algorithm proceed with the planarity test, otherwise the graph is declared non-planar. Once again a depth-first search is used to number all the vertices in the graph, and a direction is imposed to the edges, which is the direction in which they are traversed. The search also divides the graph into two classes of directed edges: a set of *tree edges*, which define a *spanning tree* T (a tree T which is a subgraph of G and T contains all the vertices of G) of G, and a set of *back edges* (v, u), such that $v, u \in T$ and there is a path from u to v in T. A graph partitioned in such way is called a *palm tree*. Figure 4.6 shows an example of graph and its palm and spanning tree. Finally the search computes two values for each vertex, the $lowpt1$ and the $lowpt2$, where $lowpt1$ and $lowpt2$ are defined as follows:

$$lowpt1(v) = \min(\ \{v\} \cup S_v\ ), \text{ and}$$
$$lowpt2(v) = \min(\ \{v\} \cup (S_v \text{ - } \{lowpt1(v)\})\ ),$$

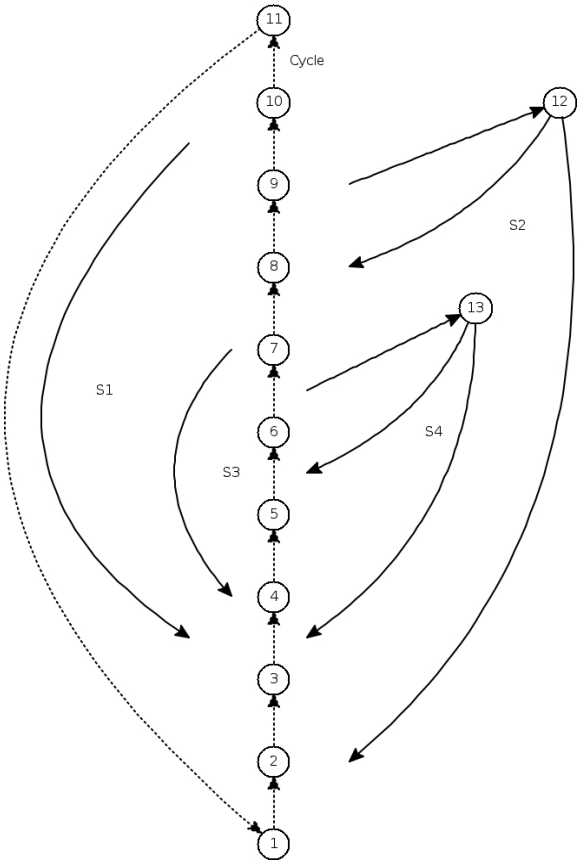
where v is a vertex, and S_v the set of vertices reached by back edges from descendants of v. The $lowpt1$ is the vertex with lowest number than v reachable by a back edge from a descendant of v, and $lowpt2$ is the second vertex with lowest number than v reachable by a back edge from a descendant of v. Then the adjacency lists are sorted according to the increasing value of $\phi((v, u))$, where ϕ is a function defined on the edges of the graph's palm tree as follows:



(a) An example graph.



(b) A palm tree generated from the example graph. The solid edges are tree arcs, and they form a spanning tree. The dotted edges are the back edges.



(c) The first generated cycle is represent with dotted edges, and the segments with the solid edges.

Figure 4.6: Some examples of the planarity notions.

$$\phi((v,u)) = \begin{cases} 2 \cdot u & \text{if } (v,u) \text{ is a back edge} \\ 2 \cdot lowpt1(u) & \text{if } (v,u) \text{ is a tree edge and } lowpt2(u) \geq v \\ 2 \cdot lowpt1(u) + 1 & \text{if } (v,u) \text{ is a tree edge and } lowpt2(u) < v \end{cases}$$

Now the Auslander, Parter & Goldstein algorithm can be applied. This algorithm finds a cycle in the graph and deletes it, leaving a set of connected components called *segments*. Then the algorithm recursively tests the planarity of each segment, and determines if the segment can be embedded on the inside (left side) of the cycle or on the outside (right side) of the cycle. A cycle is formed by a path from a vertex v to a vertex u in the spanning tree plus a back edge (u, v). All the cycles besides the first one, when needed, use part of the path of the parent cycle. The segments cycles besides the first one, may have to use part of its parent cycle path. The edges leaving the cycle are called *attachments*. The *Block* structure represents a set of embedded segments, that contains a list of the left attachments and a list of the right attachments. Blocks can be flipped in order to avoid interlacing with other Blocks.

The recursive planarity test function *IsPlanar* takes three arguments: an edge, which is an attachment of the previous cycle, and a structure where the ordered attachments of this segment will be returned. The pointer *Att* points to a list structure that is used to keep the ordered list of attachments that will be returned.

```
int IsPlanar(PE_Edge *edge, Return_Att *RAtt){
  Stack *s=NULL;
  Int_List *Att=NULL;
  Block *block=NULL;
```

The function starts by finding the segment's cycle. To do that, it starts at the target vertex of *edge* and then traverses the first edge in the adjacency list (previously sorted) of each vertex until the first back edge occurs. During this process each source vertex of the traversed edge is marked as parent of the target one, so that later the cycle can be traversed backwards during the embedding process.

```
  PE_Vertex *x=edge->source;
  PE_Vertex *y=edge->target;
```

```
y->pe->parent=x;
PE_Edge *e=y->firstedge;
while(e->pe->backedge==0){
  e->target->pe->parent=e->source;
  e=e->target->firstedge;
}
PE_Vertex *wk=e->source;
PE_Vertex *w0=e->target;
```

Now all the vertices in the cycle are traversed in backwards, and a stack s with the embedded Blocks is kept. For each edge (v, u) emanating from the cycle, the function is recursively applied, except if it is a back edge, in which case, an attachments list is created consisting only in the vertex u. Then a Block is created using the returned attachments list or the created one. This Block only has attachments on the left side. If it interlaces with the top Block of s (last block embedded), then it must be flipped (the left attachments are swapped with the right attachments) to see if it can be embedded on the right side. If it is still interlaced, the graph is declared non-planar. If that is not the case, the Block is finally tested to see if its right side interlaces with the left side of the last embedded Block. In the affirmative case, the two Blocks are combined and pushed to s, otherwise the Block is simply pushed to s. After embedding all the segments emanating from a vertex w, the attachments higher than that vertex can be cleared and definitely marked with its embedding side (left or right).

```
Return_Att RA;
RA.Att=NULL;
PE_Vertex *w=wk;
while(w->number!=x->number){
  e=w->firstedge->next;
  while(e!=NULL && e->pe->backedge>=0){
    RA.Att=NULL;
    if(e->pe->backedge==0){
r=IsPlanar(e,&RA);
    }else{
```

```
Int_List *item=(Int_List *)malloc(sizeof(Int_List));
item->value=e->target->number;
item->next=RA.Att;
RA.Att=item;
      }
      block = Block_New(e,RA.Att);
      RA.Att=NULL;
      while(s!=NULL){
if(Block_Left_Interlace(block,s))
   Block_Flip(s->topblock);
if(Block_Left_Interlace(block,s))
   return FALSE;
if(Block_Right_Interlace(block,s)){
   block=Block_Combine(block,s->topblock);
   s=Stack_Pop(s);
}
else
   break;
      }
      s=Stack_Push(s,block);
      block=NULL;
      e=e->next;
   }
   w=w->pe->parent;
   while(s!=NULL && Block_Clean(s->topblock,w->number)){
      s=Stack_Pop(s);
   }
}
```

Finally the attachments list to be returned is created using the Blocks from s, and each Block is tested to see if it interlaces the cycle's back edge, which make the graph non-planar.

```
while(s!=NULL){
   Block *tblock=s->topblock;
   if(tblock->Latt!=NULL && tblock->Ratt!=NULL &&
```

```
        tblock->Latt->value > w0->number && tblock->Ratt->value > w0->
            number){
        return FALSE;
      }
      Att=Int_List_Concat(Att,Block_2Att(tblock,w0->number));
      s=Stack_Pop(s);
    }
    if(w0->number!=x->number){
      Att=Int_List_Append(Att,w0->number);
    }
    RAtt->Att=Att;
    Att=NULL;
    return TRUE;
}
```

The function $IsPlanar$ tests if the graph is planar. If that is the case, it computes each segment side. Although it does not computes a complete embedding of the graph. To do that the function $Embedding$ traverses the graph using a depth-first search, and computes a planar embedding using the side information about each segment, provided by the function $IsPlanar$. A planar embedding is given by the final ordering of the adjacency lists.

The function $Embedding$ uses a depth-first search similar to the one used in the planarity test to recursively compute a planar embedding. A set of auxiliary lists are used to kept the clockwise ordered edges. The list T has the ordered edges that will form the adjacency list of the current vertex w. The list A is the concatenation list of AL,$(wk, w0)$ and AR, where AL and AR are the ordered lists of incident edges to the left and to the right side of the cycle, respectively. Each recursively embedded segment S returns the lists $TPrime$ and $APrime$, where $TPrime$ is the ordered list of edges incident to w embedded in S and $APrime$ is the ordered list of edges incident to the cycle embedded in S. The segment S can be embedded in the left or right side of the cycle. If S is embedded in the left side then T becomes the concatenation list of $TPrime$ and T, and AL the concatenation list of AL and $APrime$, otherwise T

becomes the concatenation list of T and $TPrime$, and AR the concatenation list of $APrime$ and AR (see Figure 4.7).

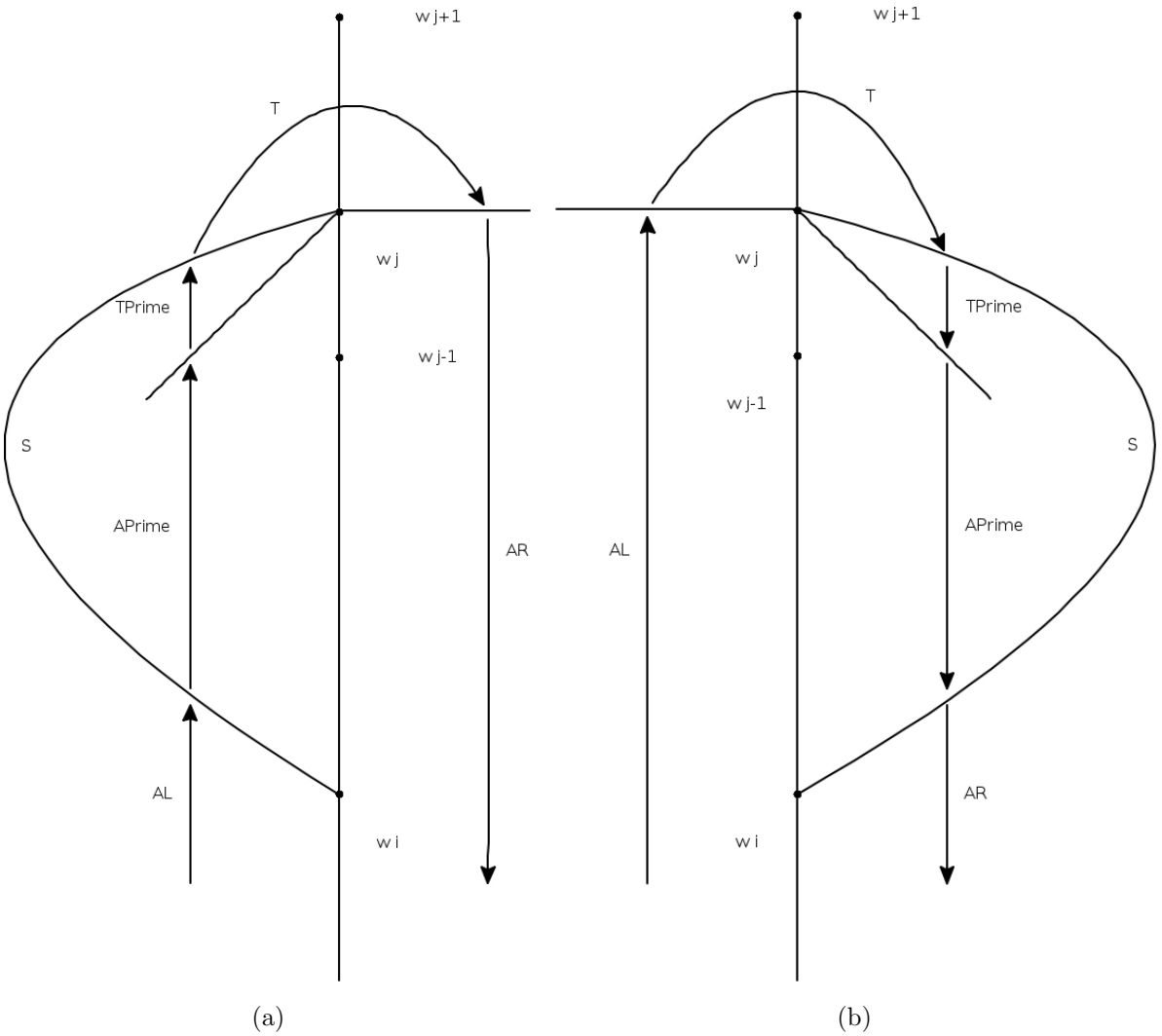


Figure 4.7: Embedding the segment S on the left and right side of the cycle, respectively.

The embedding function receives four arguments: a tree edge $e0$, the side where the segment is going to be embedded, and the lists T and A described above. This function starts by finding the cycle just as in the case of the function $IsPlanar$.

```
void Embedding(PE_Edge *e0, int side, TA_List *T, TA_List *A){
  int rv=1;
  PE_Vertex *e0_source=e0->source;
```

```
e0->target->pe->parent=e0_source;
e0->target->pe->edge_in=e0;
PE_Edge *e=e0->target->firstedge;
while(e->pe->backedge==0){
  e->target->pe->parent=e->source;
  e->target->pe->edge_in=e;
  e=e->target->firstedge;
}
PE_Vertex *wk=e->source;
PE_Edge *cycle_backedge=e;
```

The list T is initialized with the back edge $(wk, w0)$ and all the others are initialized empty.

```
PE_Edge_List *newedgeR;
PE_Edge_List *newedge=PE_Edge_List_New(cycle_backedge);
PE_Edge_List *cycle_backedge_R=PE_Edge_List_New(cycle_backedge->
    reversal);
TA_List_Append(T,newedge);
TA_List TPrime;
TA_List APrime;
TPrime.Head=TPrime.Tail=NULL;
APrime.Head=APrime.Tail=NULL;
TA_List AL;
TA_List AR;
AL.Head=AL.Tail=NULL;
AR.Head=AR.Tail=NULL;
```

The cycle is now traversed on backwards and for each edge (w, u) emanating from the cycle vertex w, an embedding is recursively computed and the lists $TPrime$ and $APrime$ return the embedding result as described above. Then the lists $TPrime$ and $APrime$ are concatenated as described also above.

```
PE_Vertex *w=wk;
while(w->number!=e0_source->number){
  e=w->firstedge->next;
  while(e!=NULL && e->pe->backedge>=0){
```

```
      TPrime.Head=TPrime.Tail=NULL;
      APrime.Head=APrime.Tail=NULL;
      if(e->pe->backedge==0){
  int pside=RIGHT;
  if(e->pe->side==side) pside=LEFT;
  Embedding(e,pside,&TPrime,&APrime);
      }else{
  newedge=PE_Edge_List_New(e);
  TA_List_Append(&TPrime,newedge);
  newedgeR=PE_Edge_List_New(e->reversal);
  TA_List_Append(&APrime,newedgeR);
      }
      if(e->pe->side==side){
  TA_List_Concat(&TPrime,T);
  TA_List_Concat(T,&TPrime);
  TA_List_Concat(&AL,&APrime);
      }else{
  TA_List_Concat(T,&TPrime);
  TA_List_Concat(&APrime,&AR);
  TA_List_Concat(&AR,&APrime);
      }
      e=e->next;
    }
```

After all the edges emanating the vertex w are embedded, the list T concatenated with the edge (w, u), where u is the parent vertex of w, becomes the adjacency list of w. The list AL is split into the lists AL' and T' and AR into the lists AR' and T'', where T' and T'' contains all the incident edges to parent vertex of w. The list T then becomes the concatenation of the lists T' and T'', AL and AR become the lists AL' and AR', respectively.

```
    newedgeR=PE_Edge_List_New(w->pe->edge_in->reversal);
    TA_List_Push(T,newedgeR);
    w->pe->planar_embedding=T->Head;
    T->Head=T->Tail=NULL;
    PE_Edge_List *ppt=NULL,*pt=AL.Head;
```

```
    while(pt!=NULL && !(pt->edge->source->id==w->pe->parent->id)){
      ppt=pt;
      pt=pt->next;
    }
    if(pt!=NULL){
      T->Head=pt;
      T->Tail=AL.Tail;
      if(ppt!=NULL){
ppt->next=NULL;
AL.Tail=ppt;
      }else{
AL.Head=AL.Tail=NULL;
      }
    }
    newedge=PE_Edge_List_New(w->pe->edge_in);
    TA_List_Append(T,newedge);
    ppt=NULL,pt=AR.Head;
    while(pt!=NULL && (pt->edge->source->id==w->pe->parent->id)){
      ppt=pt;
      pt=pt->next;
    }
    if(ppt!=NULL){
      T->Tail->next=AR.Head;
      T->Tail=ppt;
      ppt->next=NULL;
      AR.Head=pt;
      if(pt==NULL)
AR.Tail=NULL;
    }
    w=w->pe->parent;
  }
```

At the end, T contains the ordered edges emanating from source vertex of $e0$, and the list A becomes the concatenation list of AR, the cycle back edge and AL.

```
  TA_List_Concat(A,&AR);
  TA_List_Append(A, cycle_backedge_R);
  TA_List_Concat(A,&AL);
}
```

4.7 Planar Biconnectivity Augmentation

The planar biconnectivity augmentation problem consists in adding edges to a graph to make it biconnected without destroying its planarity. To do this FAgoo implements an algorithm presented by Kant [Kan93]. The original algorithm is due to Read [Rea87], but for a graph $G = (V, E)$, this algorithm could increase the degree of a single vertex by $O(n)$, with $|V| = n$. Kant modified the algorithm so that each vertex receives at most two extra incident edges. The algorithm assumes that the input graph is connected, a planar embedding is given, the vertices are sorted in a depth-first order and the neighbors of a vertex v that belong to the same biconnected component appear in a consecutive sequence in the adjacency list of v. For every vertex v of the graph, an edge (u, w) is added for every pair of vertices u and w, that appear consecutive in the adjacency list of v and belong to different biconnected components. If the edge (v, u) was previously added by this algorithm then it is removed. The same is done for the edge (v, w).

The function *MakeBiconnected* takes a planar connected graph *graph* and adds edges in order to make it biconnected. The vertices were previously sorted in a depth-first order, so that now they are traversed in such an order.

```
void MakeBiconnected(PE_Graph *graph){
  PE_Edge *newedge=NULL,*newedgeR=NULL;
  PE_Vertex *v=graph->firstvertex;
  while(v!=NULL){
    if(v->number<0)
      break;
    PE_Edge *edge=v->firstedge;
```

```
PE_Edge *edge2=edge->next;
while(edge2!=NULL){
```

If the consecutive edges *edge* and *edge*2 belong to different biconnected components, then a edge from the target vertex of *edge* to target vertex of *edge*2 is added.

```
    if(PE_EBlock_GetId(edge->block)!=PE_EBlock_GetId(edge2->block)){
PE_Edge *eitem=PE_Edge_GetPrev(edge->reversal);
PE_Edge *eitemR=edge2->reversal;
newedge = PE_Edge_New(NULL, -1,-1,edge->target,edge2->target);
newedgeR = PE_Edge_New(NULL, -1,-1,edge2->target,edge->target);
newedge->reversal=newedgeR;
newedgeR->reversal=newedge;
PE_Edge_InsertAfter(newedge,eitem);
PE_Edge_InsertAfter(newedgeR,eitemR);
PE_EBlock_Merge(edge->block,edge2->block);
newedge->block=edge2->block;
newedgeR->block=edge2->block;
newedgeR=newedge=NULL;
```

If *edge* was added in a previous step of this function then it is removed.

```
if(edge->id<0){
  PE_Edge_Remove(edge); }
```

Analogously to *edge*, if *edge*2 was added in a previous step of this function then it is removed.

```
if(edge2->id<0){
  edge=edge2;
  edge2=edge2->next;
  PE_Edge_Remove(edge);
}
    }
    edge=edge2;
    edge2=edge2->next;
  }
  v=v->next; } }
```

Figure 4.8 shows the biconnected graph as result of the application of the biconnectivity augmentation algorithm to the graph G_1 (Figure 4.3).

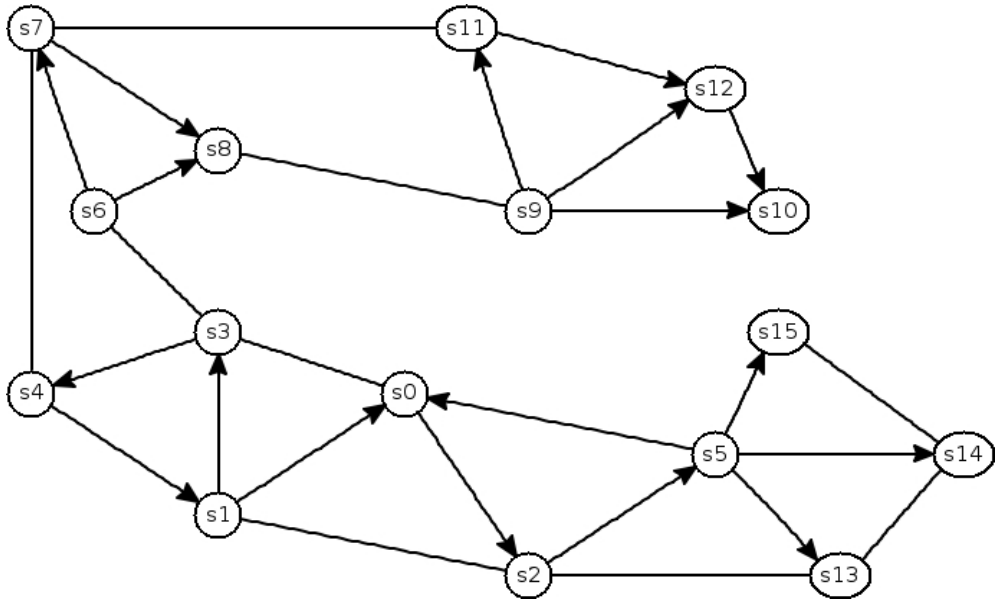


Figure 4.8: A biconnected graph.

4.8 Triangulation

A planar graph which all faces have exactly three vertices is said to be triangular. If a planar graph is not triangular then there is at least one face that has four or more vertices. A simple way to triangulate a planar graph is to add edges to the faces with more than three vertices until all faces have exactly three vertices. Note that, this approach may cause multiple edges to appear. Kant presents a simple triangulation algorithm [Kan93] that runs in linear time and as side effect it computes a canonical ordering for triangular planar graphs [dFPP90].

Given a planar graph $G = (V, E)$ and a planar embedding of G, the algorithm starts with a subgraph $G_2 = (\{v_1, v_2\}, \{(v_1, v_2)\})$ of G, and at each step it adds a vertex v_k to G_{k-1}, with $3 \leq k \leq n$ and $|V| = n$, such that G_k is triangular. An edge (u, w) with $u \in G_{k-1}$ and $w \in (G - G_{k-1})$ is called an *outgoing edge*. Let C_{k-1} be

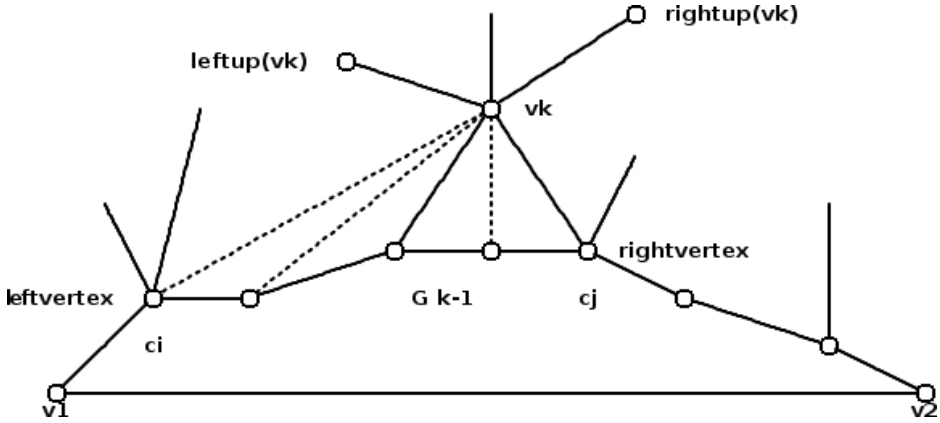


Figure 4.9: Triangulating a planar graph.

the exterior face of G_{k-1} with vertices $v_1 = c_1, c_2, ..., c_r = v_2$. The $leftvertex$ of v_k is the vertex $c_i \in C_{k-1}$, with lowest i, in the adjacency list of v_k. Analogously the $rightvertex$ of v_k is the vertex $c_j \in C_{k-1}$, with highest j, in the adjacency list of v_k. A vertex u applies to be v_k, if $u \in (G - G_{k-1})$ and all its neighbors that belong to G_{k-1} appear in a consecutive sequence on its adjacency list, i.e., there are no outgoing edges between the edges (v_k, c_j) and (v_k, c_i). The consecutive vertex $v_l \in (G - G_{k-1})$ to c_i in the adjacency list of v_k is called the $leftup$ of v_k, and the precedent vertex $v_r \in (G - G_{k-1})$ to c_j in the adjacency list of v_k is called the $rightup$ of v_k. If v_k has only one outgoing edge then $leftup = rightup$, and if it has no outgoing edges then $leftup = rightup = nil$. To control if a vertex can be the next v_k each vertex has two variables: *old* and *visit*. The variable *old* counts the number of neighbors in G_{k-1} of a vertex u, and the variable *visit* is incremented by one for every consecutive pair of neighbors of u that belong to G_{k-1}. If $old = visit + 1$ then all the neighbors of u in G_{k-1} appear consecutively in the adjacency list, thus it qualifies to be the next v_k. Such vertices are kept in a linked list called *readylist*. Some of these notions are illustrated in Figure 4.9.

The algorithm was slightly modified in order to not only compute a canonical ordering but to compute a lmc-ordering. Kant defined that *"A canonical ordering is a leftmost canonical ordering if we can add in any step k a vertex set V_k with leftvertex c_l or a vertex set $V_{k'}$ with leftvertex $c_{l'}$, and $l < l'$ holds, then $k < k'$"*. So to ensure this, the

algorithm keeps the *readylist* sorted so that any vertex u with a $leftvertex$ c_l occurs before than any vertex u' with a $leftvertex$ c'_l with $l < l'$. This is done by controlling the order that the vertices are added to the *readylist*. At each step the vertices that are tested to be added to the *readylist* are: the *rightup* of v_k's $leftvertex$ (c_i), the $leftup$ of v_k, the *rightup* of v_k and the $leftup$ of v_k's *rightvertex* (c_j).

The function *Triangulate* receives a planar biconnected graph $G = (V, E)$ and computes a leftmost canonical ordering while triangulating the graph. The vertices *old* and *visit* are initialized with 0. The ordering will be given by the vertices *number*, which is initialized with $|V| + 1$. Then two arbitrary vertices are chosen to be v_1 and v_2.

```
void Triangulate(PE_Graph *graph){
  PE_Vertex *vertex=graph->firstvertex;
  while(vertex!=NULL){
    vertex->number=graph->V+1;
    vertex=vertex->next;
  }
  PE_Vertex *v1=graph->firstvertex;
  v1->number=0;
  PE_Edge *rightup=v1->firstedge;
  PE_Edge *v1v2=rightup->next;
  PE_Edge *leftup=PE_Edge_GetNext(v1v2->reversal);
  PE_Vertex *v2=v1v2->target;
  v2->number=1;
```

The *old* variable of the neighbors of v_1 and v_2 is incremented by one.

```
  PE_Edge *edge=v1->firstedge;
  while(edge!=NULL){
    edge->target->ct->old++;
    edge=edge->next;
  }
  edge=v2->firstedge;
  while(edge!=NULL){
      edge->target->ct->old++;
```

```
      edge=edge->next;
  }
  PE_Vertex *cl=rightup->target;
  PE_Vertex *cr=leftup->target;
```

If the *rightup* of v_1 and the $leftup$ of v_2 are the same vertex then its *visit* is incremented by one and added to the ready list. Otherwise they are tested and if they qualify ($old = visit + 1$) then they are added to the *readylist*. The *readylist* works like a stack, that is why the $leftup$ of v_2 is pushed before the *rightup* of v_1.

```
  PE_Vertex_List *readylist=NULL;
  if(cl->id==cr->id){
    cl->ct->visit++;
    readylist=PE_Vertex_List_Push(readylist,cl);
    cl->ct->ready=1;
  }else{
    if(cr->ct->old==(cr->ct->visit+1)){
      readylist=PE_Vertex_List_Push(readylist,cr);
      cr->ct->ready=1;
    }
    if(cl->ct->old==(cl->ct->visit+1)){
      readylist=PE_Vertex_List_Push(readylist,cl);
      cl->ct->ready=1;
    } }
```

The external face of G_{k-1}, $outerface$, is kept as a linked list of edges, to be easy to navigate forwards and backwards on the external list. The $outerface$ is initialized with the edges (v_2, v_1) and (v_1, v_2).

```
  PE_Edge_List *outerface=NULL;
  outerface=PE_Edge_List_Push(outerface,v1v2);
  outerface=PE_Edge_List_Push(outerface,v1v2->reversal);
  PE_Vertex *vk;
  int k=2;
  while(k<graph->V){
    vk=NULL;
```

Each step starts by taking a vertex from the *readylist* to be v_k. During each step some vertices in the *readylist* may need to be removed from it, but instead of removing then, their flag *ready* is set to $false$. To find a vertex ready to be the next v_k, vertices are pop from the *readylist* until one with the *ready* flag set to *true* is found.

```
PE_Vertex_List *rl=readylist;
while(rl!=NULL && !rl->vertex->ct->ready){
  readylist=rl->next;
  rl->next=NULL;
  PE_Vertex_List_Free(rl);
  rl=readylist;
}
vk=readylist->vertex;
rl=readylist;
rl->vertex->ct->ready=0;
readylist=rl->next;
rl->next=NULL; PE_Vertex_List_Free(rl);
vk->number=k;
```

Now that v_k is found, all its neighbors *old* are incremented by one and the *ready* flag is set to $false$ for the ones that do not belong to G_{k-1} ($number > k$).

```
int deg=0;
edge=vk->firstedge;
while(edge!=NULL){
  if(edge->target->number>k){
edge->target->ct->old++;
edge->target->ct->ready=0;
  }
  deg++;
  edge=edge->next;
}
vk->ct->degree=deg;
```

The next task is to find the $leftvertex$ and $rightvertex$ of v_k (c_i and c_j, respectively). This is done by traversing the $outerface$ and testing the vertices $rightup$ and $leftup$.

The *leftvertex* is the last vertex with outgoing edges that occurs before an vertex that has an edge to v_k, and the *rightvertex* is the first vertex with outgoing edges after it.

```
    PE_Edge_List *ci=outerface;
    PE_Edge_List *cj=NULL;
    PE_Edge_List *ck=outerface;
    while(ck!=NULL){
      leftup=PE_Edge_GetNext(ck->edge->reversal);
      if(leftup->target->number>k)
ci=ck;
      while(leftup->target->number>k)
leftup=PE_Edge_GetNext(leftup);
      if(leftup->target->number==k){
cj=ck;
break;
      }
      if(ck->edge->target->ct->degree!=ck->edge->target->ct->old)
ci=ck;
      ck=ck->next;
    }
    if(ck!=NULL && ck->next!=NULL){
      leftup=PE_Edge_GetNext(ck->edge->reversal);
      while(leftup->target->number!=k)
leftup=PE_Edge_GetNext(leftup);
      leftup=PE_Edge_GetNext(leftup);
      if(leftup->target->number<k){
cj=ck->next;
ck=cj;
    while(ck!=NULL){
      if(ck->next==NULL){
cj=ck;
break;
      }
      leftup=PE_Edge_GetNext(ck->edge->reversal);
      if(leftup->target->number>k){
```

```
cj=ck;
break;
      }
      if(leftup->target->number==k){
leftup=PE_Edge_GetNext(leftup);
if(leftup->target->number>k){
   cj=ck;
   break;
}
      }
      ck=ck->next;
   } } }
```

The *leftup* of v_k is the previous edge of its first outgoing edge in the adjacency list, and the *rightup* is the first edge after the last outgoing edge in the adjacency list.

```
   PE_Edge *next=NULL;
   rightup=vk->firstedge;
   leftup=vk->firstedge;
   if(leftup->target->number<k){
      while(leftup!=NULL && leftup->target->number<k){
rightup=leftup=leftup->next;
      }
      if(leftup!=NULL){
next=rightup->next;
while(next!=NULL && next->target->number>k){
   rightup=next;
   next=next->next;
}
      }
   }else{
      next=rightup->next;
      while(next->target->number>k){
rightup=next;
next=next->next;
      }
```

```
      while(next!=NULL && next->target->number<k)
  next=next->next;
      if(next!=NULL)
  leftup=next;
    }
    PE_Edge *vkleftup=leftup;
    PE_Edge *vkrightup=rightup;
```

To triangulate G_k, edges from $c_i, ..., c_j$ to v_k are added if not present yet.

```
    PE_Edge_List *add2vk=NULL;
    PE_Edge *firstedge=NULL,*lastedge=NULL;
    ck=ci;
    while(ck!=cj->next){
      PE_Edge *prev=ck->edge->reversal;
      leftup=PE_Edge_GetNext(prev);
      while(firstedge==NULL && leftup->target->number>k){
  prev=leftup;
  leftup=PE_Edge_GetNext(leftup);
      }
      if(leftup->target->id!=vk->id){
  PE_Edge *newedge = PE_Edge_New(NULL, -1,-1,ck->edge->target,vk);
  PE_Edge *newedgeR = PE_Edge_New(NULL, -1,-1,vk,ck->edge->target);
  newedge->reversal=newedgeR;
  newedgeR->reversal=newedge;
  PE_Edge_InsertAfter(newedge,prev);
  add2vk=PE_Edge_List_Push(add2vk,newedgeR);
  if(firstedge==NULL)
    firstedge=newedge;
  lastedge=newedgeR;
      }else{
  if(firstedge==NULL)
    firstedge=leftup;
  lastedge=leftup->reversal;
  if(add2vk!=NULL){
    PE_Edge *prev=lastedge;
    while(add2vk!=NULL){
```

```
      PE_Edge_InsertAfter(add2vk->edge,prev);
      prev=add2vk->edge;
      PE_Edge_List *tofree=add2vk;
      add2vk=add2vk->next;
      tofree->next=NULL;
      PE_Edge_List_Free(tofree);
    }
  }
      }
      ck=ck->next;
    }
    if(add2vk!=NULL){
      PE_Edge *prev=vkrightup;
      if(prev==NULL)
  prev=firstedge->reversal;
      while(add2vk!=NULL){
  PE_Edge_InsertAfter(add2vk->edge,prev);
  prev=add2vk->edge;
  PE_Edge_List *tofree=add2vk;
  add2vk=add2vk->next;
  tofree->next=NULL;
  PE_Edge_List_Free(tofree);
      }
    }
```

The $firstedge$ is the edge (c_i, v_k) and the $lastedge$ is the edge (v_k, c_j). The $rightup$ of c_i is the previous edge of $firstedge$ in the adjacency list of c_i, and the $leftup$ of c_j is the edge after (c_j, v_k) in the adjacency list of c_j.

```
    rightup=PE_Edge_GetPrev(firstedge);
    leftup=PE_Edge_GetNext(lastedge->reversal);
    cl=NULL;
    cr=NULL;
    if(rightup->target->number<k)
      rightup=NULL;
    else
```

```
    cl=rightup->target;
  if(leftup->target->number<k)
    leftup=NULL;
  else
    cr=leftup->target;
```

The vertices that need to increment the *visit* by one are: the *rightup* of c_i if it is the $leftup$ of v_k, the *rightup* of the c_i if it is the $leftup$ of c_j and v_k has no outgoing edges ($old = degree$), and the $leftup$ of c_j if it is the *rightup* of v_k.

```
  PE_Edge *aux=vk->firstedge;
  while(aux!=NULL && aux->target->number<k)
    aux=aux->next;
  if(vk->ct->old==vk->ct->degree){
    if(cl!=NULL && cr!=NULL && cl->id==cr->id){
cl->ct->visit++;
    }
  }else{
    if(cl!=NULL && cl->id==vkleftup->target->id){
cl->ct->visit++;
    }
    if(cr!=NULL && cr->id==vkrightup->target->id){
cr->ct->visit++;
    }
  }
```

Now the $leftup$ of the c_j, the *rightup* of v_k, the $leftup$ of v_k and the *rightup* of c_i, are added to the *readylist* if their $old = visit + 1$. They are push to the *readylist* by that order, to ensure a lmc-ordering.

```
  if(cr!=NULL && cr->ct->old==(cr->ct->visit+1)
     && !cr->ct->ready){
    readylist=PE_Vertex_List_Push(readylist,cr);
    cr->ct->ready=1;
  }
  if(vkrightup!=NULL && vkrightup->target->ct->old==(vkrightup->target
      ->ct->visit+1)
```

```
    && !vkrightup->target->ct->ready){
  readylist=PE_Vertex_List_Push(readylist,vkrightup->target);
  vkrightup->target->ct->ready=1;
}
if(vkleftup!=NULL && vkleftup->target->ct->old==(vkleftup->target->
    ct->visit+1)
    && !vkleftup->target->ct->ready){
  readylist=PE_Vertex_List_Push(readylist,vkleftup->target);
  vkleftup->target->ct->ready=1;
}
if(cl!=NULL && cl->ct->old==(cl->ct->visit+1)
    && !cl->ct->ready){
  readylist=PE_Vertex_List_Push(readylist,cl);
  cl->ct->ready=1;
}
```

Finally the *outerface* is updated, by replacing the edges between c_i and c_j, with the $firstedge$ and *lastedge*.

```
PE_Edge_List *tofree=ci->next;
PE_Edge_List *nextl=NULL;
cj=cj->next;
while(tofree!=cj){
  nextl=tofree->next;
  tofree->next=NULL;
  PE_Edge_List_Free(tofree);
  tofree=nextl;
}
ci->next=cj;
PE_Edge_List *new_list=PE_Edge_List_Push(NULL,lastedge);
if(new_list==NULL){
  rv=-1;
  goto freeall;
}
new_list=PE_Edge_List_Push(new_list,firstedge);
```

```
if(new_list==NULL){
  rv=-1;
  goto freeall;
}
ci->next=new_list;
new_list=new_list->next;
new_list->next=cj;
k++; } }
```

Figure 4.11 is the triangulation result of the graph in Figure 4.10.

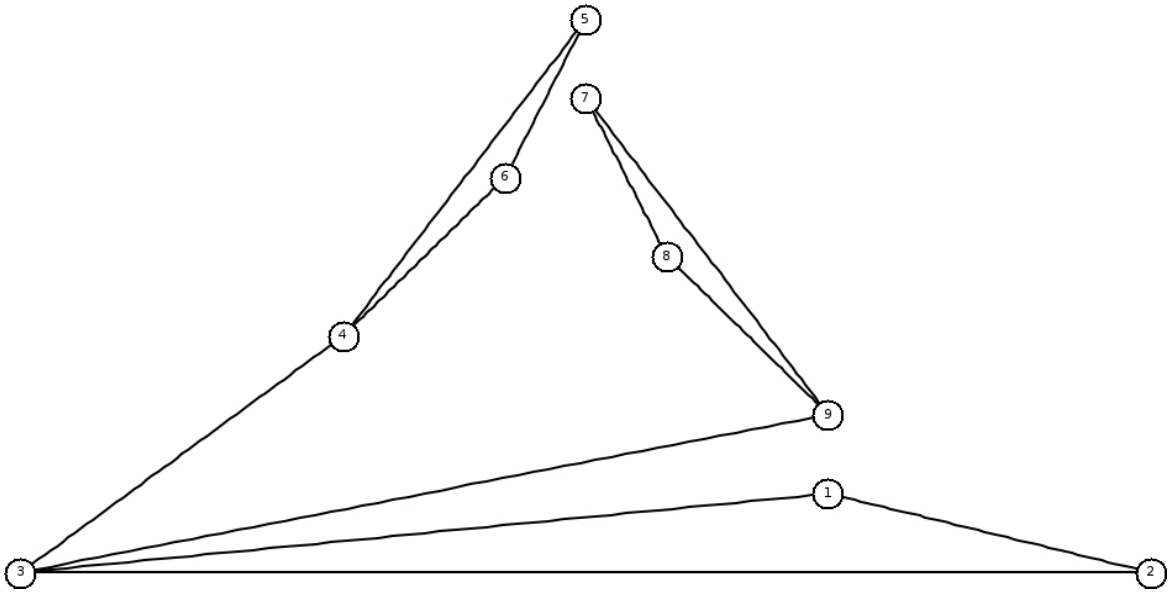


Figure 4.10: Example graph G_2.

4.9 Straight Line Drawing

The straight-line drawing algorithm implemented in FAgoo [Kan93] is originally due to Fraysseix et al. [dFPP90]. The algorithm takes as input a triangular planar graph. It also uses the lmc-ordering computed during the triangulation process.

The algorithm starts with v_1 at (0,0) and at each step k of the lmc-ordering, $x(c_1) < x(c_2) < ... < x(c_m)$, where $v_1 = c_1, c_2, ..., c_m = v_2$ is the external face of G_k. The

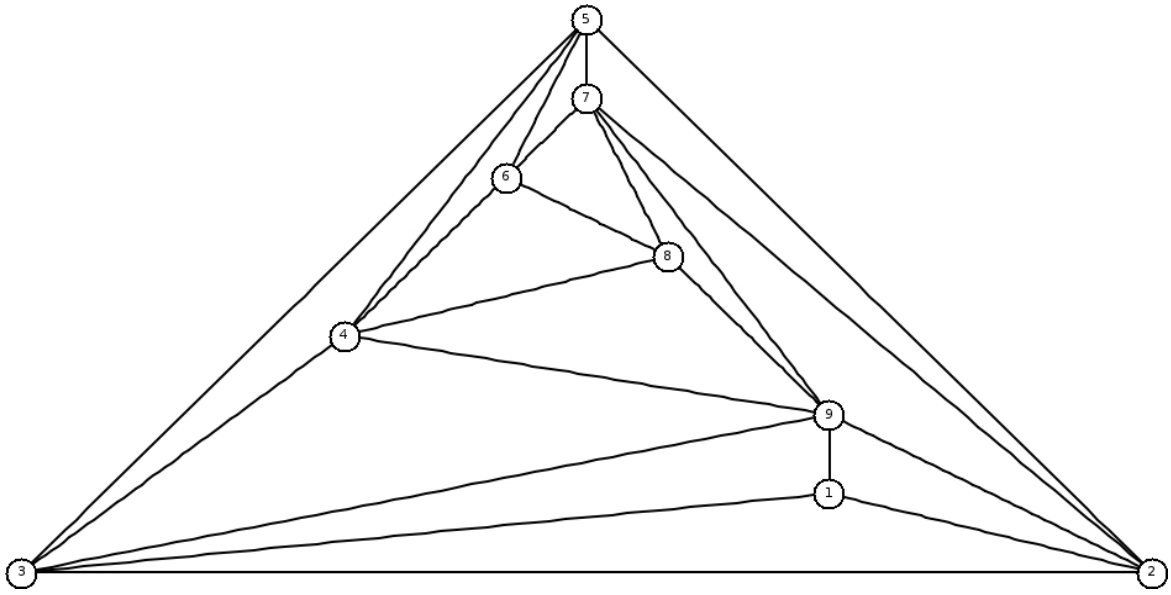


Figure 4.11: A triangulation of G_2.

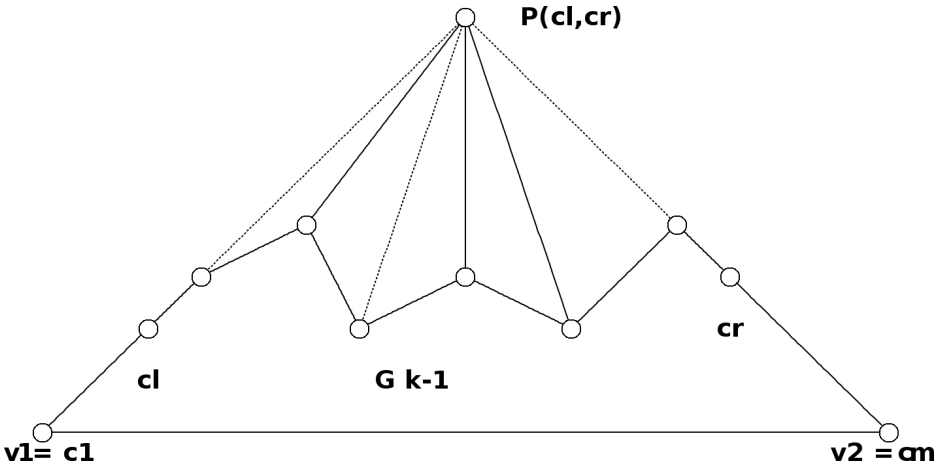
edges (v_l, v_{l+1}) have slopes of $+1$ or -1. Each time a vertex v_k, with a $leftvertex$ c_l and $rightvertex$ c_r, is added to G_{k-1}, the vertices $c_{l+1}, ..., c_{r-1}$ in the external face of G_{k-1} are shifted one position to the right, and the vertices $c_r, ..., c_m$ in the external face of G_{k-1} are shifted two position to the right. Several internal vertices of G_{k-1} must also be shifted to the right. These internal vertices will be shifted in a second stage of the algorithm. The vertex v_k is now placed at the crossing point of the line with slope $+1$ that passes through c_l and the line with slope -1 that passes through c_r. All the vertices $c_l, ..., c_r$ in the external face of G_{k-1} are visible from this point. An illustration of a step k of this algorithm is shown in Figure 4.12.

To correctly deal with the shifts applied to the vertices, a flag, *correct*, is assigned to each vertex u. This flag denotes whether $x(u)$ must be recalculated or not. There is also a counter, $shift$, for each vertex u, which count the number of shifts that must be done to the right. At each step k of the lmc-ordering, the *correct* and $shift$ of v_k are set to $false$ and to 0, respectively. Then the vertex c_j with lowest j and *correct* set to $false$ is found. For each vertex c_n with $j \leq n < r$, $x(c_n)$ is set to $\sum_{j \leq i \leq n} shift(c_i)$ and *correct* of c_n is set to $true$. After recalculating the x position of all vertices $c_j, ..., c_{r-1}$, the $\sum_{j \leq i < n} shift(c_i)$ is added to the $shift$ of c_n.

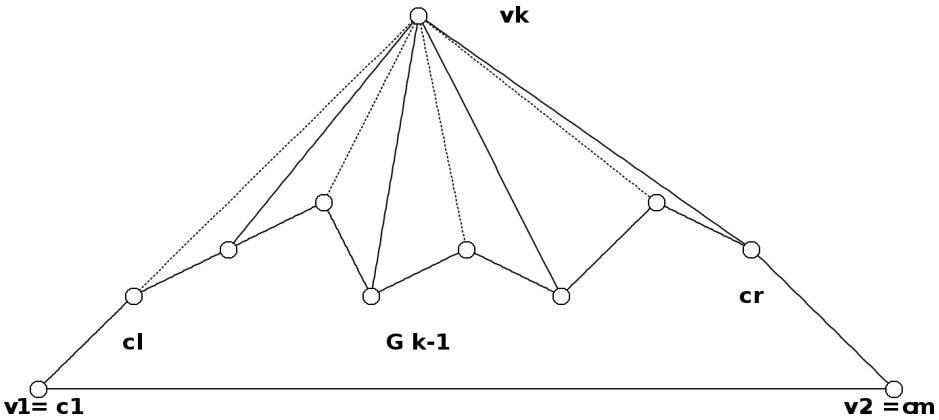
Finally, to correctly shift all the internal vertices of each step k, the algorithm traverses the vertices in the reversal ordering, and uses the $shift$ and $rshift$ counters to propagate the shifts to the internal vertices.

The function *GetStraightLineDraw* starts by setting an array with pointers to the graph's vertices with a lmc-ordering. The position of the vertices $v1$ and $v2$ were already initialized at (0,0), and the *correct* of $v1$ can be set to *true*, since its final position will be (0,0). The external face of G_{k-1} is kept in the vertex list $outerface$.

```
void GetStraightLineDraw(PE_Graph *graph){
  PE_Vertex *lmc_ordering[graph->V];
  PE_Vertex *vertex=graph->firstvertex;
```



(a)



(b)

Figure 4.12: Two examples of compound labels.

```
while(vertex!=NULL){
  lmc_ordering[vertex->number]=vertex;
  vertex=vertex->next;
}
PE_Vertex *v1=lmc_ordering[0];
PE_Vertex *v2=lmc_ordering[1];
v1->sld->correct=1;
PE_Vertex_List *outerface=NULL;
outerface=PE_Vertex_List_Push(outerface,v2);
outerface=PE_Vertex_List_Push(outerface,v1);
```

The the graph is traversed in a lmc-ordering. Each step k starts by recalculating the x value of the vertices $c_j, .., c_{r-1}$, and the $shift$ value of c_r, as described above. The variable SLD_SPACE is used to add an extra shift to the vertices and scatter more the drawing.

```
int k=2;
while(k<graph->V){
  PE_Vertex *vk=lmc_ordering[k];
  PE_Vertex *cl=vk->sld->cl->target;
  PE_Vertex *cr=vk->sld->cr->target;
  PE_Vertex_List *ci=NULL;
  PE_Vertex_List *cj=NULL;
  PE_Vertex_List *cv=outerface;
  while(cv!=NULL && cv->vertex->id!=cr->id && cv->vertex->sld->correct
      ){
    if(cv->vertex->id==cl->id)
ci=cv;
    cv=cv->next;
  }
  int sum=0;
  while(cv!=NULL && cv->vertex->id!=cr->id){
    if(cv->vertex->id==cl->id)
ci=cv;
    sum+=cv->vertex->sld->shift;
    cv->vertex->sld->x+=sum;
```

```
    cv->vertex->sld->correct=1;
    cv=cv->next;
  }
  cj=cv;
  cr->sld->shift+=sum+(2*SLD_SPACE);
```

Then the position of v_k is calculated, by intersecting the lines that pass c_l and c_r with slopes $+1$ and -1, respectively, and the *outerface* is updated.

```
  int x1,y1,b1;
  int x2,y2,b2;
  int bm;
  x1=cl->sld->x;
  y1=cl->sld->y;
  b1=y1-x1;
  x2=cr->sld->x+cr->sld->shift;
  y2=cr->sld->y;
  b2=y2+x2;
  bm=b2-b1;
  if(bm!=0)
    bm/=2;
  vk->sld->x=bm;
  vk->sld->y=b1+bm;
  PE_Vertex_List *tofree=ci->next;
  PE_Vertex_List *nextl=NULL;
  while(tofree!=cj){
    nextl=tofree->next;
    tofree->next=NULL;
    PE_Vertex_List_Free(tofree);
    tofree=nextl;
  }
  ci->next=cj;
  cj=PE_Vertex_List_Push(cj,vk);
  if(cj==NULL){
    rv=-1;
    goto freeall;
```

```
    }
    ci->next=cj;
    k++;
  }
```

The second stage of the function starts by resetting the $shift$ value of the vertices to 0. Then the vertices are traversed in the reversal order. The $shift$ and $rshift$ of v_k are added to the $shift$ of the vertices $c_{l+1}, ..., c_{r-1}$. This propagates the shifts applied to v_k to its internal vertices. The $rshift$ is also added to the vertex c_r. The v_k's final x position in now set by adding it the $shift$ value.

```
int i;
for(i=0;i<graph->V;i++)
  lmc_ordering[i]->sld->shift=0;
k--;
while(k>1){
  PE_Vertex *vk=lmc_ordering[k];
  PE_Edge *vi=PE_Edge_GetPrev(vk->sld->cl);
  while(vi->target->id!=vk->sld->cr->target->id){
    vi->target->sld->shift=vk->sld->shift+vk->sld->rshift+SLD_SPACE;
    vi=PE_Edge_GetPrev(vi);
  }
  vk->sld->cr->target->sld->rshift+=vk->sld->rshift+(2*SLD_SPACE);
  vk->sld->x+=vk->sld->shift;
  k--;
}
v2->sld->x+=v2->sld->shift+v2->sld->rshift;}
```

Figure 4.10, Figure 4.13, and Figure 4.14 are some examples of straight-line drawing created using this algorithm.

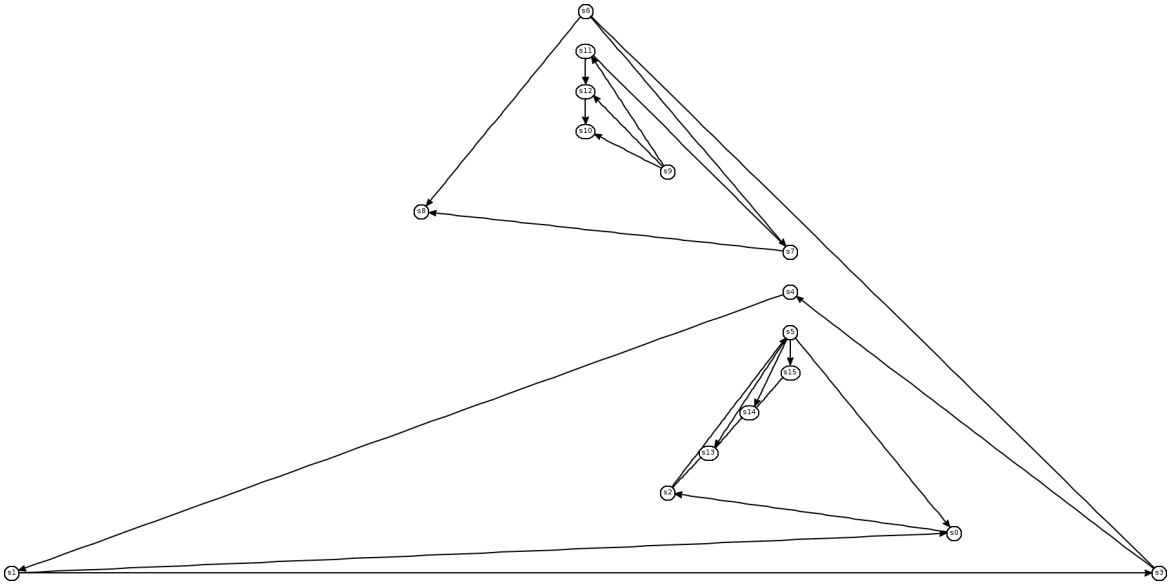


Figure 4.13: A straight-line drawing of G_2 computed by FAgoo.

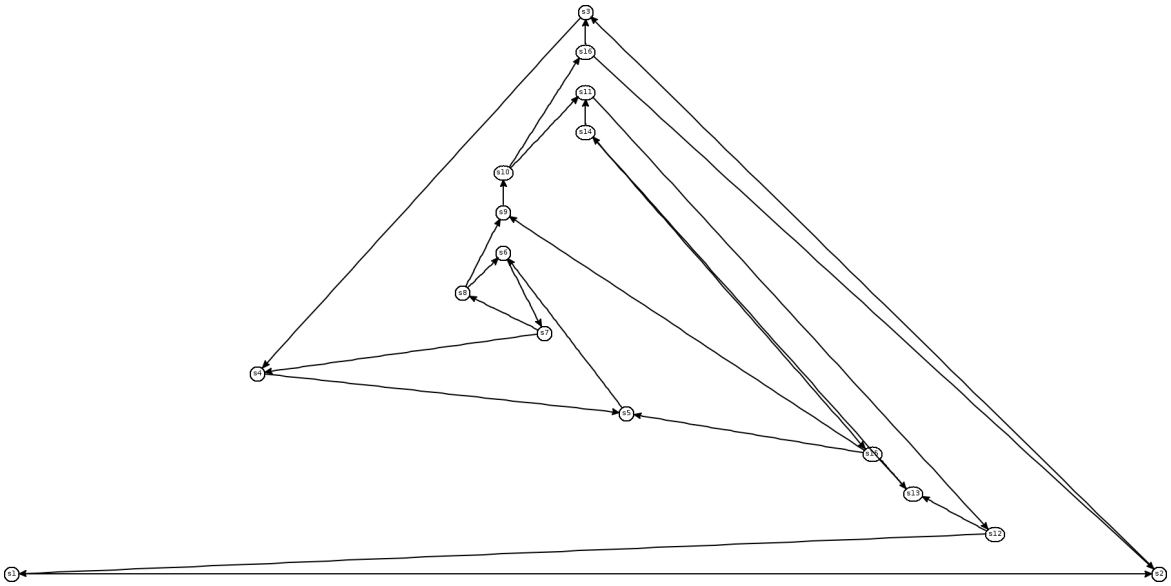


Figure 4.14: A straight-line drawing of G_3 computed by FAgoo.

Chapter 5

Conclusions

This thesis shows that finite automata drawings have very specific drawing conventions that usually can not be achieved by simply applying the already existent graph drawing algorithms. Some applications and libraries for graph and finite automata drawing and visualization are presented. The overall appreciation of these softwares in finite automata drawings is weak, which is comprehensible since most of them were not specifically designed to draw finite automata.

A new graphical environment, GUItar, was developed for the visualization, editing and manipulation of finite automata diagrams. This software allows the creation of rich node and arc styles for diagram drawings widening its scope to other types of diagrams. It provides a powerful mechanism, the FFCs, that allows the easy integration of other software libraries, such as FAdo and FAgoo. With this mechanism, GUItar pretends to converge several software libraries and increase the potential of each one.

FAgoo is the first work of what intends to be a library for finite automata drawings. This software was implemented as a Python module written in C because this way it combines a Python high level interface, allowing its integration in GUItar, with the performance efficiency of the C programming language. The Table 5.1 shows a summary of the implemented algorithms in FAgoo.

Function	Input Graph	Output
$BDFS$	Connected graph	Computes the graph biconnected components and its BC-tree.
$IsPlanar$	Biconnected graph	Return $true$ if the graph is planar, $false$ otherwise.
$Embedding$	Biconnected planar graph	Computes a planar embedding of the graph.
$MakeBiconnected$	Connected planar graph and a planar embedding	Biconnected planar graph.
$Triangulate$	Biconnected planar graph and a planar embedding	Triangular planar graph.
$GetStraightLineDraw$	Triangular planar graph	Computes a planar straight-line drawing.

Table 5.1: A summary of the implemented algorithms.

As for future work, we intend to improve GUItar and implement some new features such as the combining of nodes and edges, the collapsing of subgraphs, and the animation of algorithms. We also intend to continue the implementation of FAgoo, which still needs to implement many other graph drawing algorithms, as well as the development and adaptation of graph drawing algorithms for finite automata diagrams. We also pretend to implement an interactive force-directed model in FAgoo.

Bibliography

[AAA+09] André Almeida, Marco Almeida, José Alves, Nelma Moreira, and Rogério Reis. Fado and guitar: tools for automata manipulation and visualization. In S. Maneth, editor, *CIAA 2009: Fourteenth International Conference on Implementation and Application of Automata*, volume 5642 of *LNCS*, pages 65–74. Springer-Verlag, 2009.

[aiS10] aiSee Graph Layout Software. aiSee. `http://www.aisee.com/`, Access date:28.06.2010.

[AMR10] José Alves, Nelma Moreira, and Rogério Reis. XML description for automata manipulations. In Alberto Simões, Daniela Cruz, and José Carlos Ramalho, editors, *Actas XATA 2010, XML: aplicações e tecnologias associadas*, pages 77–88, ESEIG, Vila do Conde, 2010.

[AP61] L. Auslander and S.V. Parter. On embedding graphs in the plane. *J. Math. Mech.*, 11:517–523, 1961.

[Bar10] C. Barker. Floatcanvas. `http://trac.paulmcnett.com/floatcanvas`, Access date:28.06.2010.

[BBL95] Paola Bertolazzi, Giuseppe Di Battista, and Giuseppe Liotta. Parametric graph drawing. *IEEE Trans. Software Eng.*, 21(8):662–673, 1995.

[Boo10] Boost C++ Libraries. Boost.python. http://www.boost.org/doc/libs/1_43_0/libs/python/doc/index.html, Access date:28.06.2010.

[Dav10] Dave Beazley and SWIG developers. Swig. http://www.swig.org/, Access date:28.06.2010.

[dFPP90] Hubert de Fraysseix, János Pach, and Richard Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[FAd10a] FAdo Project. GUItar: tools for diagrams visualization. http://www.ncc.up.pt/FAdo, Access date:28.06.2010.

[FAd10b] FAdo Project. FAdo: tools for formal languages manipulation. http://www.ncc.up.pt/FAdo, Access date:29.03.2010.

[Fou10] Python Software Foundation. Python language website. http://python.org, Access date:28.06.2010.

[Gol63] A. J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in the plane. In *Graph and Combinatorics Conf., Contract No. NONR 1858-(21), OOEce of Naval Research Logistics Proj., Dep. of Math. Princeton U., May 1618*, pages pp,, 1963.

[Gra10a] Graph Visualization Software. The dot language. http://www.graphviz.org, Access date:28.06.2010.

[Gra10b] GraphML Working Group. The GraphML file format. http://graphml.graphdrawing.org, Access date:28.06.2010.

[H. 10] H. de Fraysseix and P. Ossona de Mendez. P.I.G.A.L.E. - Public Implementation of a Graph Algorithm Library and Editor. http://pigale.sourceforge.net/, Access date:28.06.2010.

[Har69] Frank Harary. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969.

[HT73] John E. Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation [h] (algorithm 447). *Commun. ACM*, 16(6):372–378, 1973.

[HT74] John E. Hopcroft and Robert Endre Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[jgr10] jgraph. jgraph. `http://www.jgraph.com/`, Access date:28.06.2010.

[Kan93] Goossen Kant. *Algorithms for Drawing Planar Graphs*. PhD thesis, Universiteit Utrecht, Faculteit Wiskunde en Informatica, 1993.

[Kur30] Kazimierz Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:271–283, 1930.

[LC10] AT&T Research Labs and Contributors. Graphviz. `http://www.graphviz.org`, Access date:28.06.2010.

[LS09] S. Lombardy and J. Sakarovitch. Vaucanson-G. `http://igm.univ-mlv.fr/~lombardy`, Access date:1.12.2009.

[MM96] Kurt Mehlhorn and Petra Mutzel. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.

[Rea87] R. C. Read. A new method for drawing a planar graph given the cyclic order of the edges at each vertex. *Congressus Numerantium*, (56):31–44, 1987.

[RF06] Susan H. Rodger and Thomas W. Finley. *JFLAP: An interactive formal languages and automata package*. Jones & Bartlett Publishers, 2006.

[Tec10] Technical University of Dortmund and oreas GmbH. OGDF - Open Graph Drawing Framework. `http://www.ogdf.net/`, Access date:28.06.2010.

[yWo10] yWorks GmbH. yWorks. `http://www.yworks.com/`, Access date:28.06.2010.